Beyond "Protected" and "Private": An Empirical Security Analysis of Custom Function Modifiers in Smart Contracts

Yuzhou Fang*
The Hong Kong University of Science and Technology
Hong Kong SAR, China
yzfang@cse.ust.hk

Daoyuan Wu†‡
The Chinese University of Hong Kong
Hong Kong SAR, China
dywu@ie.cuhk.edu.hk

Xiao Yi
The Chinese University of Hong Kong
Hong Kong SAR, China
yx019@ie.cuhk.edu.hk

Shuai Wang†
The Hong Kong University of Science and Technology
Hong Kong SAR, China
shuaiw@cse.ust.hk

Yufan Chen
Xidian University
Xi'an, China
fanwatcher144@gmail.com

Mengjie Chen
Mask Network
Shanghai, China
rachel_chen0915@outlook.com

Yang Liu‡
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

Lingxiao Jiang
Singapore Management University
Singapore, Singapore
lxjiang@smu.edu.sg

ABSTRACT

A smart contract is a piece of application-layer code running on blockchain ledgers and it provides programmatic logic via transaction-based execution of pre-defined functions. Smart contract functions are by default invokable by any party. To safeguard them, the mainstream smart contract language, i.e., Solidity of the popular Ethereum blockchain, proposed a unique language-level keyword called "modifier," which allows developers to define custom function access control policies beyond the traditional "protected" and "private" modifiers in classic programming languages.

In this paper, we aim to conduct a large-scale security analysis of the modifiers used in real-world Ethereum smart contracts. To achieve this, we design and implement a novel smart contract analysis tool called SoMo. Its main objective is to identify insecure modifiers that can be *bypassed* from one or more unprotected smart contract functions. This is challenging because of the complicated relationship between modifiers and their variables/functions and the ambiguity of attacker-accessible entry functions. To overcome them, we first propose a new structure, the *Modifier Dependency Graph (MDG)*, to connect all the modifier-related control/data flows. Over MDGs, we then model system variables, generate symbolic path constraints, and iteratively test each candidate entry function. Our extensive evaluation shows that SoMo outperforms the state-of-the-art SPCon tool by detecting all its true positives and correctly avoiding 9 out of 11 false positives. It also achieves high precision of 91.2% when analyzing a large dataset of 62,464 contracts, over 400 of which were identified with bypassable modifiers. Our analysis further reveals three interesting security findings about modifiers and nine major types of modifier usage in the wild. SoMo has been integrated into an online security scanning service, MetaScan.

1 INTRODUCTION

The rise of blockchain technology has lead to the immense development of decentralized applications [15, 54]. Ethereum [55] has gained significant attention in recent years due to its unique capacity to facilitate running Turing-complete smart contracts for various purposes. However, vulnerabilities in smart contracts have emerged as a significant threat to the whole Ethereum ecosystem [12, 22, 24, 31, 37, 40, 44, 48, 52, 58]. If smart contracts contain flaws, the inability to change deployed contracts or reverse transactions once they have been written to the blockchain can have severe consequences. Given that smart contract computations often involve cryptocurrency transactions, the potential for financial loss is particularly high. For instance, the infamous Dao attack [2] resulted in the theft of over 3.6 million Ethers (the cryptocurrency of Ethereum), causing a mandatory hard fork of Ethereum.

Ethereum smart contracts are written in a high-level language called Solidity [8], the bytecode of which runs in stack-based virtual machine named Ethereum Virtual Machine (EVM) [4]. As smart contracts are designed to interact with Ethereum transactions, Solidity functions are by default invokable by any party unless they are specified with the `internal` or `private` visibility specifiers. To allow developers to enforce more flexible function access control policies beyond the traditional "protected" and "private" modifiers in classic programming languages, Solidity proposed a unique language-level keyword called "modifier." Specifically, a modifier declares a piece of conditional checks that are automatically embedded by Solidity into the function prologues during contract compilation. Modifier conditions often check against the global or contract-wide state

*Half of the work by Yuzhou Fang was done at the Chinese University of Hong Kong.
†Daoyuan Wu and Shuai Wang are the corresponding authors.
‡Daoyuan Wu and Yang Liu are also affiliated with MetaTrust Labs during this study.

ISSTA 2023, 17-21 July, 2023, Seattle, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

variables[1] and system variables[2]. However, state variables might be manipulated by attackers from an unprotected function, causing a bypass of the corresponding modifiers and further affecting the functions protected by insecure modifiers. The infamous Parity Wallet bug [3] that led to over 150,000 Ethers stolen was actually due to this reason, and we will illustrate the Parity bug in §2.

In this paper, we conduct an empirical security analysis of custom function modifiers in real-world Ethereum smart contracts. Our main objective is to identify insecure modifiers that could be *bypassed* from one or more unprotected smart contract functions. We refer to such modifiers as *bypassable modifiers* in this paper. To automatically detect them, however, requires us to address two unique challenges on modifier analysis due to (i) mutual dependability between modifiers, variables used by modifiers, conditions related, and functions that change the variables and (ii) the ambiguity of entry functions, whereby any function — even those with modifiers — could become an attacker-accessible entry function.

We propose a novel smart contract analysis tool called SoMo to address the above challenges. Specifically, SoMo first performs backward slicing to construct a structure called *modifier dependency graph (MDG)*, which leverages global state variables to *connect* all the modifier-related control and data flows. On top of the generated MDG, SoMo then *iteratively* explores sliced paths for different modifiers and tests their path feasibility from each candidate entry function by modelling system variables and resolving symbolic path constraints. Besides detecting bypassable modifiers, SoMo also collects all the modifiers to allow a modifier-specific NLP (Natural Language Processing) analysis for understanding their major usage in the wild.

To evaluate SoMo, we first benchmark it with a state-of-the-art tool with similar objectives. The most related to our work are SPCon [37] and Ethainter [13], both of which explicitly consider the impact of modifiers in their analysis. Specifically, Ethainter [13] takes into account modifier-related sanitization during its Datalog-based information flow analysis. On the other hand, SPCon [37] identifies privileged functions that should not be accessible to normal users by analyzing the transaction history. Since SPCon targets at a problem much closer to ours, we benchmark SoMo with SPCon using its public dataset and found that among the 34 contracts with modifiers, SoMo can detect all the 23 true positives while correctly avoiding 9 out of 11 false positives made by SPCon.

We further conduct a large-scale experiment with 62,464 real-world smart contracts collected from multiple sources, including 5,000 recent contracts that were submitted to Etherscan [5] on 2 November 2022. Overall, SoMo reports a total of 500 vulnerable contracts and marks 61,481 contracts secure[3]. We cross-check all the 500 vulnerable contracts reported and confirm that 456 of them are true positives, including 411 actually vulnerable and the other 45 with modifiers intentionally designed to be bypassable under certain conditions (e.g., a transaction with a certain amount of tokens). As a result, SoMo achieves good precision at 91.2%, which suggests that SoMo is practically accurate to analyze a large set of real-world smart contracts. Moreover, SoMo is quite fast, with 95% of the 62,464 contracts finished within 2 seconds.

SoMo's results also reveal three interesting security findings, including (i) SoMo identified more percentage of vulnerable contracts with Ether than that in previous works, which demonstrates the importance of the problem on bypassable modifiers that may not be realized by developers; (ii) earlier versions of Solidity caused some developers to mistakenly expose constructor functions as the public entry functions, causing around 80% of the vulnerable contracts in our dataset; and (iii) the default behavior of Solidity to set the functions with no visibility specifiers public may not match with developers' understanding, leading to another 20% of the vulnerable contracts. Besides these insights about bypassable modifiers, our NLP analysis and clustering categorize major modifier usage into four categories —*access control*, *financial related*, *contract state*, and *misc check* and summarize nine types of common modifiers defined in the wild.

Contributions. To sum up, we make the following major contributions in this paper:

- *(Methodology)* We propose a novel tool SoMo to detect bypassable modifiers, in which we (i) design a *modifier dependency graph* (MDG) to correlate modifier-related control- and data-flows with state variables and (ii) *iteratively* and *symbolically* explore and test each candidate entry function.
- *(Evaluation)* We conduct extensive evaluation by comparing SoMo with the state-of-the-art SPCon tool and applying it to analyze a large dataset of 62,464 real-world smart contracts. The result shows high accuracy and efficiency of SoMo.
- *(Insights)* Our results reveal three interesting findings about the root causes of modifier-related vulnerabilities and nine types of major modifier usage in the wild.

Availability. SoMo has been integrated as a part of MetaScan[4], an industry-leading smart contract security scanning platform. To facilitate future comparisons with SoMo, we have made the dataset publicly available on https://github.com/VPRLab/ModifierDataset.

Roadmap. The rest of this paper is organized as follows. We first provide some background about Solidity and its modifiers in §2. We then present SoMo's design and evaluation in §3 and §4, respectively. Furthermore, we discuss some threats to internal and external validity in §5 and summarize related work in §6. Lastly, we conclude the paper in §7.

2 BACKGROUND

In this section, we introduce some background about Solidity and its modifiers, as well as the security implication of modifiers.

Solidity. Ethereum offers a stack-based virtual machine named Ethereum Virtual Machine (EVM) [4] for executing the bytecode compiled from smart contract programs written in a high-level language called Solidity [8]. Solidity provides four visibility specifiers, namely `public`, `external`, `internal`, and `private`, to determine the accessibility to contract components (i.e., functions and variables). While `private` and `internal` specifiers prevent potentially untrusted parties from accessing contract elements, `public` and

[1]State variables in Solidity refer to the non-volatile variables that are stored in the EVM storage space and have a persistent impact on the contract.
[2]System variables refer to special variables like `msg.sender` preserved in the global namespace of Solidity.
[3]There are also 483 contracts failed. We will explain the details in §4.2.

[4]https://metatrust.io/metascan

```
1  pragma solidity ^0.4.9;
2  contract WalletLibrary {
3      address owner;
4      modifier onlyowner(){
5          require(msg.sender == owner);
6          _;
7      }
8      function initWallet(address _owner){
9          // Should only be called by constructor
10         owner = _owner;
11         // ... more setup ...
12     }
13     function execute(...) external onlyowner{
14         // execute any transaction by owner immediately
15     }
16 }
```

Figure 1: The trimmed Parity Wallet bug.

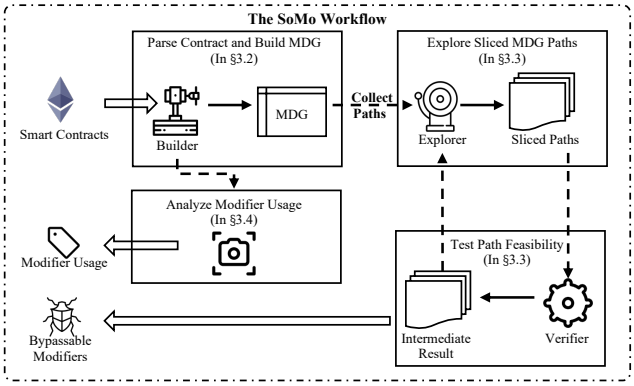

Figure 2: The overall workflow of SoMo.

`external` allow such accesses, though `external` prohibits the call from other functions in current contract.

Modifiers. Function modifiers are a smart contract language feature used to modify the behavior of general functions. Modifiers can be used to enforce specific conditions that must be met prior to executing a function, such as checking the sender's identity or the amount of the Ether appended. By using modifiers, developers can create reusable and modular code that can be easily applied to multiple functions, ensuring consistent behavior across the contract. This includes, but is not limited to, defining a modifier to enforce sensitive functions like `mintTokens` that are accessible to a specific group of users. Typically, modifiers are designed as preconditions to check properties before executing contract functions. Therefore, they are often associated with the privileges of contracts.

The security implication. Due to the access control nature of contract modifiers, improper usage can lead to unintended and severe consequences. For example, Figure 1 shows the trimmed code of the infamous Parity Wallet bug [3], which resulted in the loss of millions of dollars. The root cause of this bug is a bypassable modifier `onlyowner`, which was exploitable via the unintentionally exposed function `initWallet`. Specifically, attackers can first invoke function `initWallet` to update the state variable `owner` such that the included check (line 5 in Figure 1) in `onlyowner` is satisfied. With the bypassed modifier `onlyowner`, attackers can further invoke the function `execute` to drain funds from the contract. To fix such a severe vulnerability, developers should set the `initWallet` function `private` or `internal` to prevent unintended exposure.

3 THE SOMO TOOL

To enable an empirical security analysis of modifiers, we design a novel smart contract analysis tool called SoMo. Its main objective is to identify insecure modifiers that could be bypassed from one or more unprotected smart contract functions. As mentioned in §1, we refer to such modifiers as *bypassable modifiers* in this paper. Most of them lead to access control vulnerabilities and are thus vulnerable, but we also observe that a few of them are intentionally designed by developers so that they can be triggered under certain conditions (e.g., a transaction with a certain amount of tokens). SoMo does not aim to distinguish these two in its automated analysis because both cases are technically bypassable. One human analyst can easily differentiate them from SoMo's output by understanding the context of a smart contract.

Besides detecting bypassable modifiers, SoMo also collects all the modifiers to allow an NLP (Natural Language Processing) analysis of these modifiers for understanding their usage in the wild. In this section, we present the design towards the main objective from §3.1 to §3.3, and describe the part of NLP analysis in §3.4.

3.1 Overview and Challenges

In this section, we first give an overview of SoMo's design and implementation, then illustrate its major challenges using a running example contract.

Overview. Figure 2 presents the overall workflow of SoMo. It has four major components. First, the *builder* component takes an input contract source code, parses it to generate intermediate representation (IR), and generates a modifier dependency graph (MDG) via backward slicing. Second, on top of the generated MDG, the *explorer* component collects and iteratively explores sliced paths for different modifiers. Third, the *verifier* component symbolically validates the path feasibility of existing sliced paths and presents the intermediate analysis result, i.e., some bypassable modifiers, back to the *explorer* component. The latter continues to collect new sliced paths because some entry functions protected with bypassable modifiers are now shown as invokable by adversaries. After these two steps, SoMo reports the final result of all the bypassable modifiers. Lastly, we design and apply an NLP-based method to cluster the major usage of different modifiers in the wild.

SoMo utilizes several tools for its implementation. First, SoMo relies on Slither [21] (version 0.8.2) to compile smart contract source code and provide support of a single static-assignment (SSA [47]) form IR. We choose to input contract source code instead of bytecode because the modifier information will be embedded into functions and is indistinguishable with the conditional statements in bytecode. Second, on top of the Slither SSA IR, we implement a symbolic execution prototype for Solidity by employing Z3 solver [11] (version 4.11.2) to resolve the constraints. Third, to handle different versions of smart contracts, SoMo leverages Sol-select [6] (version 0.8.2) to switch between different Solidity compilers. Overall, SoMo is implemented with 5,400 lines of Python code.

```
1  pragma solidity ^0.8.0;
2  contract Example{
3      address owner;
4      address newOwner;
5      mapping (address=>uint) balance;
6      mapping(address=>bool) admin;
7      modifier onlyOwner() {
8          address caller = msg.sender;
9          require(caller == owner);
10         _;
11     }
12     modifier onlyAdmin() {
13         require(admin[msg.sender]);
14         _;
15     }
16     function changeOwner(address _addr) public {
17         newOwner = _addr;
18     }
19     function transferOwnership() external {
20         require(msg.sender == newOwner);
21         owner = newOwner;
22     }
23     function addAdmin(address _addr) onlyOwner public{
24         admin[_addr] = true;
25     }
26     function reward(address _addr, uint _value)
           onlyAdmin public {
27         // add more tokens to the target address
28         balance[_addr] += _value;
29     }
30     function superChangeOwner(address _addr) public{
31         // A hard-coded address of developers
32         require(msg.sender == 0xABC);
33         owner = _addr;
34     }
35 }
```

Figure 3: A running example to illustrate the two main challenges that are addressed by SoMo in §3.2 and §3.3; see its corresponding modifier dependency graph in Figure 4.

Challenges. We design a running example contract in Figure 3 to illustrate the two main challenges in the course of designing SoMo. This contract defines two modifiers, `onlyOwner` and `onlyAdmin`, and use them to protect the function `addAdmin()` and `reward()`, respectively. We now explain the challenges related to these two modifiers as follows.

First, to bypass the check of modifier `onlyOwner`, an attacker needs to modify the value of the state variable `owner` (line 9) so that it is equal to the attacker's wallet address (i.e., `msg.sender` in line 8). The variable `owner` could be set to a new value (line 21) by a public function called `transferOwnership()`, but such change is only allowed when a condition (line 20) related to another variable `newOwner` is satisfied. As such, we need to further analyze the variable `newOwner` (line 17) in another function `changeOwner()`. We can see that this process involves the complicated relationship between multiple parties, i.e., modifiers, variables used by modifiers, conditions with related variables, and functions that change the variables. It is thus *challenging to identify, correlate, and resolve such relationship*, and we address this problem by proposing a new structure called *modifier dependency graph* (MDG), which summarizes all the data and control flow information required to analyze the modifiers in a contract. We present how to construct MDG in §3.2.

Second, to further bypass another modifier `onlyAdmin`, an attacker needs to similarly modify the value of a state variable `admin` (line 13). This variable could be modified by the function `addAdmin()`, but it is protected by the modifier `onlyOwner`. While we have known `onlyOwner` is bypassable through the analysis above, SoMo does not if it analyzes `onlyAdmin` first. This problem is caused by the challenge that *any function — even those with modifiers — could become an attacker-accessible entry function*. To address this challenge, SoMo must iteratively analyze different modifiers and resolve the ambiguity of a target entry function (e.g., `addAdmin()` here) by symbolically verifying whether its modifier (i.e., `onlyOwner` here) is bypassable or not. We explain how to achieve this in §3.3.

Additionally, to effectively understand the usage of a large amount of different modifiers, we need a domain-specific NLP analysis method for modifiers. We introduce our approach in §3.4.

3.2 Constructing Modifier Dependency Graph

We design modifier dependency graph (MDG) as a variant of other dependency or slicing graphs, such as code property graph (CPG) [59] and backward slicing graph (BSG) [56, 57]. Essentially, MDG is a modifier-related slicing of the traditional control- and data-flow graphs, capturing all the information required to analyze the modifiers in a smart contract.

Figure 4 shows a major portion of MDG generated for the running example contract in Figure 3. Compared to other dependency graphs, we can see that there are two unique designs in MDG. First, we treat modifiers in a way similar to functions so that SoMo can build control and data flow graphs for modifiers. This also allows to build the caller and callee relationship between functions and modifiers, such as function `addAdmin()` calls modifier `onlyOwner` via a special node of `MODIFIER_CALL` in Slither. Second, we leverage global state variables to connect different modifiers and functions. For example, while function `transferownership()` and modifier `onlyOwner` have no direct call relationship according to the traditional dependency graph, but because of smart contract's transaction mechanism, a transaction can call function `transferOwnership()` to change the value of variable `owner` and further affect modifier `onlyOwner`. We thus intentionally establish the connection edge from modifier `onlyOwner` to state variable `owner` (i.e., edge ❷ in Figure 4) and the edge from variable `owner` to its assignment statement in function `transferOwnership()` (i.e., edge ❸). In this way, MDG recovers the hidden relationship between modifier `onlyOwner` and function `transferownership()`.

Based on the inter-procedural control flow graph (I-CFG) constructed in the pre-processing stage, SoMo generates and expands MDG along with the analysis of different modifiers. It is worth noting that Slither provides only *intra*-procedural control flow graph for each single function and modifier, and SoMo thus leverages them and the call graph information to customize a complete I-CFG for the entire contract. In this I-CFG and subsequent MDG, each node is expressed in Slither SSA IR (static single assignment intermediate representation) for convenient dataflow analysis.

We now use the process of analyzing modifier `onlyOwner` to illustrate how to construct MDG. As shown in Figure 4, there are seven steps as follows:

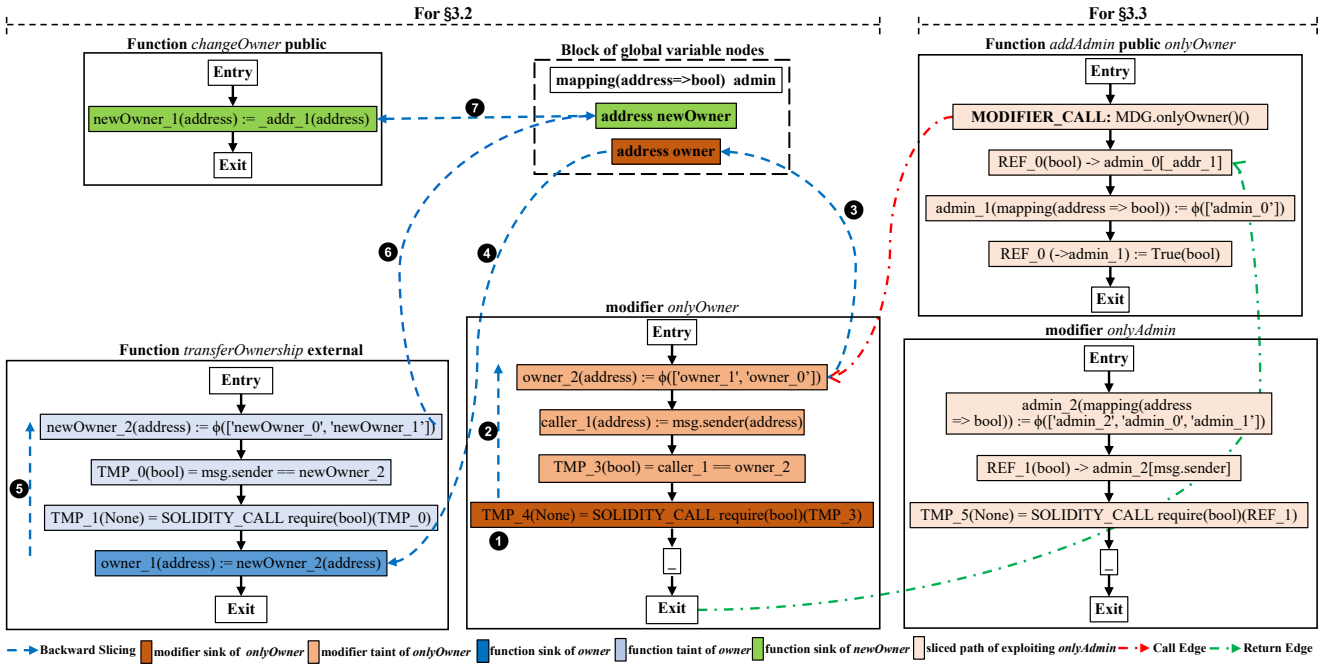

Figure 4: A major portion of the modifier dependency graph (MDG) generated for the running example contract in Figure 3. Note that here we skip the edges from modifier `onlyAdmin` to variable `admin` and further to function `addAdmin` for simplicity.

(1) Given a modifier like `onlyOwner`, the first step is to identify its *modifier sinks*, which are conditional statements used in this modifier, such as line 9 for modifier `onlyOwner` in Figure 3. Besides the `require` statement, we also cover other conditional statements, including the `if/else` and `assert` statements. In each modifier sink statement, SoMo extracts the involved tainted object (e.g., the temporary SSA variable `TMP_3` in Figure 4) and uses it as the starting point for backward slicing or taint analysis.

(2) From the tainted object `TMP_3`, SoMo performs backward slicing until reaching the modifier entry and records all the tainted statements into MDG. We call such relevant statements *modifier taints* in Figure 4. During the taint propagation, SoMo applies the *transfer function* that taints the variables on the left-hand side of assignment operations when there is an operand on the operation's right-hand side that is tainted. Meanwhile, the transfer function over-approximately taints the composite variable if one of its fields has been tainted. For instance, when analyzing the statement in line 13 that relies the on the `msg.sender` field of variable `admin` (denoted as the object `admin.msg.sender`), SoMo over-approximately marks variable `admin` as tainted instead of just tainting the object `admin.msg.sender`. This allows easier taint analysis on MDG while the potential over-estimates could be excluded during the forward symbolic testing in §3.3.

(3) At the entry of modifier `onlyOwner`, SoMo needs to propagate the tainted state variables to their block of global variable nodes and add the connection edge ❸ in Figure 4.

(4) Likewise, this step connects edge ❹ from variable `owner` in the block of global variable nodes to its assignment statement in function `transferOwnership`. This requires SoMo to identify all the assignment statements of variable `owner`, and we call such statements *function sinks* in Figure 4.

(5) Similar to step 1, this step first extracts the tainted SSA variable `owner_1` from the sink statement in function `transferOwnership`. It then performs backward slicing in a way similar to step 2 and tracks all the tainted *function taint* statements until reaching the function entry.

(6) Similar to step 3, this step makes another connection edge from the tainted state variables in function `transferOwnership` to their block of global variable nodes.

(7) Lastly, SoMo repeats the procedure above and finishes the backward slicing at function `changeOwner` because it has no further caller flows. At this stage, MDG for modifier `onlyOwner` has been successfully generated.

3.3 Iteratively Exploring Sliced Paths in MDG and Symbolically Testing Path Feasibility

With the generated MDG, SoMo iteratively explores sliced paths and symbolically tests their path feasibility. As explained previously in §3.1, the major challenge of this task is to handle the ambiguity of entry functions, whereby any function — even those with modifiers — could become an attacker-accessible entry function. As a result, SoMo must iteratively analyze different modifiers until all attacker-accessible entry functions are determined.

Take Figure 4 as an example, there are two modifiers, `onlyOwner` and `onlyAdmin`. Initially, SoMo constructs MDG for both of them and finds that the entry of `onlyOwner`'s sliced path is function

changeOwner() while the entry of onlyAdmin's sliced path is function addAdmin(). The former is certainly an entry function because it is public and can be invoked by adversaries, whereas the latter is protected by modifier onlyOwner. Hence, SoMo first analyzes modifier onlyOwner, determines its attack feasibility via symbolic execution, then further analyzes modifier onlyAdmin because function addAdmin() now becomes a callable entry function. In contrast, if a modifier is not bypassable, the functions it protects are thus not attacker-accessible entry functions.

Technically, the major difficulty is how to symbolically handle Slither IR code on the sliced paths, generate path constraints in forward execution on MDG, and resolve those constraints to test path feasibility. To do that, SoMo first maintains a collection of symbolic variables to record the path execution states and introduces any symbolic variable if it is required by the IR code semantics. Then, SoMo symbolically interprets the commonly used types of IR operations and records their corresponding constraints. After executing all code on a sliced path, SoMo invokes Z3 solver to resolve all the constraints recorded. Based on the output of Z3's constraint solving, SoMo identifies bypassable or secure modifiers. Specially, if the Z3 solver returns "unsat," indicating it is impossible to find solutions for the constraints of a path, SoMo considers this path and its corresonding modifier secure. Otherwise, if the Z3 solver can generate concrete solutions, SoMo considers the corresponding modifer bypassable.

However, not all constraints could be well handled by Z3. We take function superChangeOwner in Figure 3 an example, where the developer hard-codes her wallet address into the contract so that only herself can invoke this function to change the contract owner. When SoMo leverages Z3 to resolve the constraints, Z3 tells SoMo that there is a vulnerability in function superChangeOwner when the attacker wallet address is equal to the one in the contract, namely msg.sender == 0xABC. However, an adversary could never satisfy such a constraint because attackers cannot manipulate the developer's wallet address (unless the private key leaks), and function superChangeOwner is thus secure.

Inspired by this case, we model Solidity system variables [5] and integrate them into SoMo's constraint resolving. Specifically, we first search all the global variables reserved by Ethereum from the Solidity document [7] and retrieve the variables that are not under users' control, e.g., block's time block.timestamp and block's height block.number. We consider such variables *immutable*, i.e., they cannot be manipulated by attackers. We categorize all the immutable variables into three categories: (i) address-based, including msg.sender, tx.origin, and this; (ii) time-based, including now and block.timestamp; and (iii) related to blockchain properties, including block.basefee, block.chainid, block.coinbase, block.difficulty, block.gaslimit, and block.number. We found that by modeling these system variables, SoMo avoids a number of false positives in the real-world vulnerability detection.

3.4 NLP Analysis and Clustering of Modifiers

Besides detecting bypassable modifiers, we also aim to understand modifier usage in the wild. Typically, a modifier's name reflects its purpose and usage. For instance, the commonly used onlyOwner modifier ensures that only the contract owner can call a protected function. However, simply clustering modifier names alone does not produce effective clusters regarding modifier usage, as the actual usage is embedded within the semantic information of the modifier. Moreover, a modifier name is composed of multiple tokens, such as "only" and "owner" in the case of the onlyOwner modifier. Therefore, understanding the meaning of each token becomes essential for extracting the semantic information of a modifier. As a result, we propose a modifier-specific NLP analysis method that first analyzes the modifier tokens before conducting clustering on the modifiers.

Tokenizing modifier names and categorizing the tokenized words. Typically, developers define modifiers in a camel case style, e.g., "only", "Fee", and "Controller" for onlyFeeController. However, simply splitting the string by capitalized letters may encounter errors. For instance, onlyCEOOrOwner will be mistakenly split into "only", "CEOOr", and "Owner". To address this problem, we use Princeton WordNet [10] as our dictionary and a divide-and-conquer algorithm [1] to handle the tokens that can be fully extracted as English words. As such, onlyCEOOrOwner can be correctly tokenized into "only", "CEO", "Or", and "Owner". Eventually, we obtain 3,173 different word tokens after tokenization. We then clean them to remove the tokens appearing only once and the useless tokens. Specifically, an useless token include (i) the token is less or equal to two letters, e.g., "an" and "or", (ii) the token belongs to the stop words, e.g., "the" and "that", and (iii) the token is not in the dictionary, e.g., misspelled words and made-up words. For the remaining 1,309 tokens, we calculate a pair-wise similarity matrix according to their *semantic* similarity using the NLP analysis via spaCy [9]. Finally, we cluster the similarity matrix by Affinity Propagation [23] since it does not require pre-setting the number of clusters and performs well with a gradual tuning of the damping factor to 0.86. Under this setting, we eventually obtain a total of 75 token clusters.

Clustering modifiers according to token clusters. After tokenizing modifier names in the last step, we then assign the clusters of tokens to modifiers. Specifically, for a modifier m consisting of p tokens (i.e., $m = t_1 t_2 ... t_p$), if token t_i belongs to cluster c_j, we assign c_j to m, i.e., $m = \{c_1, c_2, ..., c_q\}$. For example, onlyOwnerOrAdmin is tokenized into {"only","owner","or","admin"}. Two tokens, "only" and "or", are removed during the cleaning process; the other tokens, "owner" and "admin", belong to token clusters "21" and "37", respectively. Hence, we assign "21" and "37" to onlyOwnerOrAdmin, i.e., onlyOwnerOrAdmin = $\{21, 37\}$. As such, we can measure the similarity between two modifiers by their Jaccard index, i.e., $\text{SIMILARITY}(m_1, m_2) = \frac{|m_1 \cap m_2|}{|m_1 \cup m_2|}$. For instance, the similarity between onlyOwner = $\{21\}$ and onlyOwnerOrAdmin = $\{21, 37\}$ is 0.5. The last step is to calculate the pair-wise similarity matrix of all the modifiers by Jaccard index and cluster the matrix into 300 clusters via Affinity Propagation (damp = 0.88).

4 EVALUATION

In this section, we aim to conduct a comprehensive evaluation of SoMo by answering the following four research questions (RQs):

RQ1: How *effective* is SoMo when being compared with the state-of-the-art tool targeting at similar problems?

RQ2: How *accurate* and *fast* is SoMo in analyzing a large set of real-world smart contracts?

[5]Note that all system variables are assigned symbolic values before they are used.

Table 1: Benchmarking SoMo with 44 vulnerable contracts reported by SPCon, a permission bug detection tool.

Contracts	SPCon		SoMo					Sum
	TP	FP	TP	TN	FP	FN	Fail	
All	31	13	23	9	-	10[◇]	2	44
Modifiers	23	11	23	9	-	-	2	34

[◇] As SoMo targets at only modifier-related vulnerabilities, 10 contracts without any modifier are thus skipped.

RQ3: What are the major *modifier usage* in the wild?
RQ4: Can SoMo identify some interesting *security findings*?

4.1 RQ1: Comparing SoMo with SPCon

We first try to benchmark SoMo with a state-of-the-art tool with similar objectives. While there are many smart contract security analysis tools [12, 13, 32, 37, 40, 48, 52], most of them are designed for generic pattern-based vulnerabilities, such as reentrancy [46], stealing Ether or frozen funds [32, 52], and transaction order dependency bugs [12]. The most related to our work are SPCon [37] and Ethainter [13], both of which explicitly consider the impact of modifiers in their analysis. Specifically, Ethainter [13] takes into account modifier-related sanitization during its Datalog-based information flow analysis. On the other hand, SPCon [37] identifies privileged functions that should not be accessible to normal users by analyzing the transaction history. Specifically, it dynamically mines the past transaction records of a contract, characterizes the roles of different users, and detects permission bugs that violate the mined security policy. Since SPCon targets at a problem much closer to ours, we benchmark SoMo with SPCon in this paper. Moreover, SPCon provided a dataset of 44 vulnerable contracts reported by its tool on GitHub[6], allowing an effective comparison.

We thus employ SoMo to analyze those 44 contracts. Table 1 shows the comparison results between SPCon and SoMo. It is worth noting that among the 44 contracts, ten of them do not contain any modifier and are thus skipped by SoMo. Therefore, we list the modifier-only result in Table 1 too. For SPCon's result, since SPCon did not provide specific labels (true or false positive, i.e., TP or FP) for each of its deemed "vulnerable" contract, we manually validate and label all the vulnerable contracts it reports. We can see that SPCon has 13 false positives, 11 of which are from the contracts with modifiers. In contrast, SoMo has no false positive for all the 34 contracts with modifiers. SoMo correctly marks 9 contracts as secure (i.e., true negative or TN), but it also fails in the remaining two contracts due to the parsing error in Slither. Overall, SoMo is better than SPCon to detect insecure modifiers in terms of introducing much fewer false positives.

The reason why SPCon could have more false positives is mainly because it relies on past transactions to infer the permissions of different contract functions. However, existing transaction records may not reveal all scenarios. It is possible that one function was accessed only by privileged users in the transaction records, but it is actually designed to be open all to users, leading to a false positive.

[6]https://github.com/Franklinliu/SpCon-Artifact/tree/master/ISSTA2022Result/SmartBugsWildResults/spcon-smartbugs

Answer to RQ1: The benchmark result shows that SoMo outperforms the state-of-the-art SPCon tool in detecting insecure modifiers. Out of the 34 contracts that contain modifiers, SoMo can detect all 23 true positives while effectively avoiding 9 out of the 11 false positives generated by SPCon.

4.2 RQ2: Applying SoMo to Real Contracts

To comprehensively measure SoMo's accuracy and performance, we further conduct a large-scale experiment with real-world smart contracts collected from multiple sources.

Dataset. We first construct our dataset by leveraging an existing contract dataset from a state-of-the-art work. Specifically, we fetched smart contracts from the open-source dataset of Sailfish [12], which includes 89,853 contracts that were uploaded to GitHub[7] on 18 February 2022. Due to the outdated nature of Sailfish's dataset, we further collected 5,000 recent contracts that were submitted to Etherscan [5] on 2 November 2022. After collecting contracts from the two sources, we exclude 32,389 contracts without modifiers. That said, 62,464 (or 66%) smart contracts contain at least one modifier in their contracts, suggesting the pervasiveness of modifier usage in smart contracts. Since our objective is to study modifiers, this paper uses those 62,464 smart contracts with modifiers as the dataset, which is available on https://github.com/VPRLab/ModifierDataset.

Result overview. We run all of our following experiments on an Ubuntu 18.04 server with 256GB of RAM and Intel Xeon E5-2683 CPU. After analyzing the large-scale dataset above, SoMo reports a total of 500 vulnerable contracts and marks 61,481 contracts secure, as shown in Table 2. There are also 483 contracts failed: 170 of them root from Slither's parsing errors (same as the two failed cases in §4.1) and the rest 313 are due to "timeout" (we configure SoMo to analyze each contract for at most 2 minutes). Since the failed 483 cases are only 0.7% of the entire dataset, we thus skip them in this paper.

Accuracy evaluation. To evaluate SoMo's results, we validate all the reported 500 vulnerable contracts and randomly sample 100 secure contracts with Ether in the smart contract balance. To guarantee the correctness of our validation, two of the authors cross-validate all the 600 contracts; when a validation conflict arise, the third author joins to resolve it. Table 2 shows the detailed validation results. We can see that for the 500 vulnerable contracts reported, we confirm 456 of them are true positives, including 411 actually vulnerable and the other 45 with modifiers intentionally designed to be bypassable under certain conditions. We refer to them as "Intended" in Table 2. We will explain such cases in the following paragraphs. We then calculate the precision, recall, and F1 score of SoMo according to $\frac{TP}{TP+FP}$, $\frac{TP}{TP+FN}$, and $\frac{2\times TP}{2\times TP+FP+FN}$, respectively. Hence, SoMo achieves good precision at 91.2%, recall at 1.0, and F1 score at 95.4%. Note that all the sampled secure contracts are actually secure, thus the number of TN is 100. The evaluation result suggests that SoMo is practically accurate to analyze a large set of real-world smart contracts.

We first analyze and discuss the root causes of those 45 contracts with intentionally bypassable modifiers. Figure 5 shows an example

[7]https://github.com/ucsb-seclab/sailfish/tree/master/data

Table 2: SoMo's results of analyzing real-world contracts.

Overall Results									
Total #		Secure			Vulnerable			Panic	
62,464		61,481			500			483	
Sample Validation									
Sample #		TP			FP			TN	Sum
Sec.	Vul.	Vulnerable	Intended	Sum	IM*	IF◇	Sum	100	600
100	500	411	45	456	39	5	44		

IM* represents the false positives caused by imprecise analysis (IM).
IF◇ represents the false positives due to implementation flaws (IF).

```
pragma solidity ^0.4.2;
contract onlyKeyHolder{
 mapping(address => uint) public balances;
 modifier onlyKeyHolders() {
  require(balances[msg.sender] >= TOKEN_NUM);
  _;
 }
 function transfer(address to, uint value){
  balances[msg.sender]=balances[msg.sender]-value;
  balances[to] = balances[to] + value;
 }
 function startAuction() onlyKeyHolders() {
    // hold some tokens to start auction.
    ...
 }
}
```

Figure 5: A bypassable true positive case. It is intentionally designed to be bypassed when certain conditions are satisfied.

```
pragma solidity ^0.8.0;
contract Approve{
    mapping (address => mapping (address => bool))
         private approved;
    modifier onlyApproved(address user) {
        require(approved[user][msg.sender]);
        _;
    }
    function approve(address spender) external {
        approved[msg.sender][spender] = true;
    }
}
```

Figure 6: A false-positive contract due to the imprecision of variable filed analysis.

of such smart contract. We can see that modifier `onlyKeyHolder` checks if a user holds enough tokens by querying the state variable `balances`. If a user is eligible to `onlyKeyHolder`, they can initiate an auction via function `startAuction()`. However, SoMo identified a bypassable path in a normal function `transfer()` that is for moving tokens between users. This is because SoMo detects that the sensitive state variable `balances` is changed after calling `transfer()`, resulting in a bypass of modifier `onlyKeyHolder`. While this is considered as a vulnerability from the technical perspective, we believe that developers intentionally design such bypassable modifiers. Therefore, we treat this detection result as a true positive, but clarify that the involved modifier is secure and is intentionally bypassable by developers.

We then explore the root cause of SoMo's 44 false positives. As listed in Table 2, the majority of false positives (39/44) come from SoMo's current coarse-grained data-flow analysis of composite

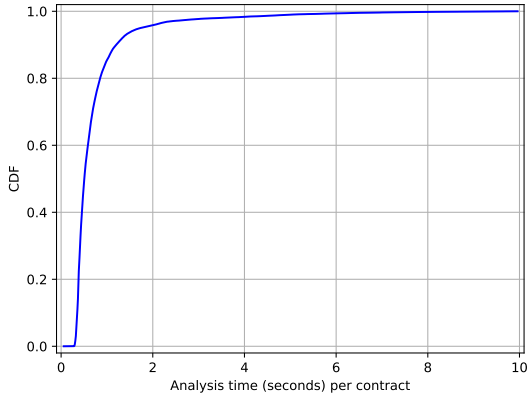

Figure 7: The CDF plot of analysis time per contract.

variables, such as `mapping`, `array`, `struct`, and etc. The remaining five false positives (5/44) are due to the implementation errors in the current SoMo prototype. We refer to them as *imprecise analysis* (IM) and *implementation flaws* (IF), respectively. Since IF-type false positives are easy to fix, we focus on the more error-prone IM-type false positives here. Figure 6 shows an example taken from the real-world contracts. This contract has a modifier `onlyApproved` that utilizes a nested mapping variable `approved` to store the approval status of different users. Users can also call function `approve()` to manage their own approval list. When analyzing this contract, SoMo mistakenly considers `approve()` exploitable as it allows users to write to the variable `approved`, which would cause modifier `onlyApproved` to be bypassed. However, each user can only access and update their own approval list, and `onlyApproved` modifier is thus secure. As a result, we know that the coarse-grained data-flow analysis of composite variables in the current SoMo prototype leads to the false positives.

Running performance. Besides accuracy results, we also measure the analysis time of each contract required by SoMo. Figure 7 presents a cumulative distribution function (CDF) plot of the analysis time per contract in SoMo. We can see that over 95% contracts were finished by SoMo with 2 seconds each and around 80% contracts need less than one second. This suggests that SoMo is quite fast for the majority of the contracts. Moreover, we find that only 1.67% (1,044 out of all the 62,464 contracts) took over 10 seconds. Analyzing 313 out of these contracts fails due to analysis timeout. A major reason is that SoMo may encounter potential graph circles in MDG, causing dead loops. We thus set a timeout to terminate such analyses.

> **Answer to RQ2:** SoMo achieves high accuracy and efficiency in analyzing real-world modifiers within a large dataset, with precision at 91.2% and able to analyze 95% of the 62,464 contracts within 2 seconds. Furthermore, SoMo identified over 400 vulnerable contracts that contain bypassable modifiers.

4.3 RQ3: Understanding Modifier Usage

Based on the NLP analysis and clustering of modifier usage in §3.4, we obtain 300 clusters for the 6,299 *unique* modifiers. Note that while our dataset consists of 134,692 modifiers in total, many of

Table 3: Nine types of major modifier usage that are summarized from our clustering result for 6,299 *unique* modifiers.

Category	ID	Type	Usage	Examples	# Modifiers
Access Control	U1	Role	Check the role of users	`onlyOwner`, `onlyAdmin`, `onlyCEO`, `onlyAuthorized`	919
	U2	Address	Check the address of users	`onlyWhitelisted`, `notBlocklisted`, `isAppoved`	322
Financial Related	U3	Transfer	Validate the transfer state	`transferLock`, `transferAuthorized`, `hasNotPaid`	266
	U4	Amount	Validate the amount of assets	`onlyIfEnoughFunds`, `adjustPrice`, `enoughFundsToPay`	252
	U5	Token	Check the token value or state	`hasTokens`, `checkMintValue`, `onlyMintable`, `tokenDoesNotExist`	222
Contract State	U6	Status	Checking contract's running status	`isActive`, `afterStart`, `isLock`, `whenNotFreeze`, `isOpenToPublic`	1033
	U7	Event	Specify events when running contracts	`isInitialized`, `onlyRecovery`, `onlyEmergency`, `onlyICO`	270
Misc Check	U8	Delegate	Verify calls from proxy contracts	`onlyDelegateFrom`, `isDelegated`, `ifDelegate`	291
	U9	General Check	Different kinds of modifier checks	`antiSpam`, `checkLocking`, `checkFihished`, `notNull`, `validUint8`	420
Sum (%)	-	-	-	-	**3,995 (63.4%)**

them share the the same names. For example, `onlyOwner`, the most common modifier name, appeared 48,027 times, and `whenPaused` and `whenNotPaused` rank as the second and third most prevalent ones, appearing 6,653 and 6,308 times respectively.

By manually checking the cluster result, we obtain the primary uses of modifiers and summarize them in Table 3. We first categorize them into four categories, namely *access control*, *financial related*, *contract state*, and *misc check*. We then present the detailed major usage and introduce corresponding examples. As shown in Table 3, our modifier usage analysis result has covered 63.4% of all the unique modifiers.

- The *access control* category covers the widely used modifiers that specify the access control policies of protected functions. Specifically, usage U1 checks the role of users through modifiers such as `onlyOwner` and `onlyAdmin`. Similarly, usage U2 checks whether a user address is in the whitelist or approved, such as `onlyWhitelisted` and `isApproved`.
- The *financial-related* category contains three types of usage, namely U3 for validating the transfer state, U4 for checking the number of assets, and U5 for validating the token value or state. For instance, the `transferLock` modifier prevents data race when transferring sensitive resources between users (U3), while the `onlyifEnoughFunds` modifier ensures that the amount of assets appended in a transaction is sufficient (U4). Similarly, in usage U5, the `hasToken` and `onlyMintable` modifiers validate whether the caller has token balances or is eligible to mint, respectively.
- The *contract state* category checks the contract state for maintaining contract running status (U6) and specifies certain contract events (U7). In particular, we find that modifiers like `onlyFrozen` and `afterStartTime` manage the contract status, allowing users to interact only during a specific period (U6). Likewise, in usage U7, the `isInitialized` modifier specifies the event of contract initialization, and the `onlyICO` modifier specifies that it is only for the ICO event.
- The last *misc check* category checks different aspects of properties and is hard to be categorized. In this category, one usage is to check whether function calls are from proxy contracts by defining modifiers like `isDelegated` (U8). This is important for developing upgradable contracts. For other checks, we categorize them into usage U9, including the `antiSpam` modifier for preventing malicious transactions.

In the future, we plan to further categorize and link the usage of modifiers with the previously summarized categories of smart contract vulnerabilities [27].

Answer to RQ3: We classify the primary usage of modifiers into four categories: *access control*, *financial-related*, *contract state*, and *misc check*. We also summarize nine types of common modifiers that are defined in real-world contracts.

4.4 RQ4: Security Findings and Case Studies

In this section, we report three major security findings obtained during vulnerability analysis of SoMo's results. We also conduct a case study of a high-value vulnerable contract with 37 Ether. For ethic considerations, we do not disclose the details of the vulnerable contracts reported in terms of their name, address, compiling arguments, and etc. We also tried to contact developers for vulnerability mitigation, but due to the decentralized and anonymous nature of smart contracts, we currently are not able to approach them. Similar situation was also encountered by the teEther team [32].

Finding 1: SoMo identified more percentage of vulnerable contracts with Ether than that in previous works. According to Perez et al. [45], many vulnerable contracts detected by six state-of-the-art techniques [26, 31, 32, 40, 44, 52] are exploitable, but hold only very limited Ether (the cryptocurrency of Ethereum). For example, Perez et al. [45] reported that *"the percentage of Ether exploited is an order of magnitude lower, with at most 0.4% of the Ether at stake exploited for re-entrancy."* As a result, the vulnerability impact or consequence was considered low. This is because when developers become aware of vulnerabilities in their contracts, they have to deploy a new contract due to the immutability of smart contracts. As a result, many vulnerable yet inactive contracts without a big threat of financial loss could still remain on Ethereum. In this paper, SoMo identified more percentage of vulnerable contracts with Ether than the above reported. Specifically, out of the 411 bypassable contracts detected, 20 or 4.87% were found to have Ether, including two contracts with over two and 37 Ether on the balance. This demonstrates the real-world value of SoMo and the importance of the problem on bypassable modifiers that may not be realized by developers.

Finding 2: Earlier versions of Solidity caused some developers to mistakenly expose constructor functions as the public entry functions. Prior to Solidity version 0.4.22, constructors were defined using a function with the same name as the contract. However, we found that this way of declaring constructors could result

```
1  pragma solidity ^0.4.20;
2  contract Owned {
3      address owner;
4      modifier onlyOwner() {
5          require(msg.sender == owner);
6          _;
7      }
8      function owned(){
9          owner = msg.sender;
10     }
11 }
```

Figure 8: An example vulnerable contract related to Finding 2.

```
1  pragma solidity ^0.4.13;
2  contract MyCrowdSale {
3      address public owner;
4      modifier onlyOwner() {
5          require(msg.sender == owner);
6          _;
7      }
8      function kill() onlyOwner {
9          selfdestruct(owner);
10     }
11     function Crowdsale(address _t,uint _s){
12         require(_t != 0);
13         require(_s != 0);
14         // anyone can become owner!
15         owner = msg.sender;
16         token = ZiberToken(_t);
17         startsAt = _s;
18     }
19 }
```

Figure 9: A high-value vulnerable contract with 37 Ether.

in unintended bugs, particularly those related to contract modifiers. Figure 8 illustrates a common mistake made by developers when using earlier versions of Solidity in defining constructors. Since Solidity is case-sensitive, the compiler does not recognize function `owned()` in Figure 8 as a constructor; instead, the compiler treats it as a normal public function. As a result, the vulnerable function `owned()` can be used to update the state variable `owner`, allowing attackers to bypass modifier `onlyOwner`. We found that around 80% of the vulnerable contracts SoMo identified were more or less due to this root cause. To address this issue, Solidity introduced the keyword `constructor` to prevent incorrect usage of contract constructors in recent versions.

Finding 3: The default behavior of Solidity to set the functions with no visibility specifiers public may not match with developers' understanding. Another major reason for modifier-related vulnerabilities is the lack of proper understanding of the function visibility. Specifically, as a permissionless and decentralized platform, Ethereum allows all users to interact with Solidity `public` and `external` functions (with no modifier) in a contract. However, unlike traditional programs, this feature could sometimes cause unintentional bugs or vulnerabilities. From the earlier example in Figure 1, developers failed to specify visibility of function `initWallet()`, and Solidity compiler by default assigns the function `public` visibility. This default behavior allows any user to invoke the function and to become the `owner` despite that the developer's intention to use it internally. We found that around 20% of the vulnerable contracts were due to this reason.

Case study. We now conduct a case study to illustrate a vulnerable contract detected with over 37 Ether, as shown in Figure 9. This contract has a modifier called `onlyOwner` (line 4), which verifies the transaction caller against the state variable `owner` (line 5). Unfortunately, this contract also includes a vulnerable function `Crowdsale()` (line 14), which allows attackers to input random data and become the contract `owner` (line 15). After becoming the owner, attackers can call `kill()` function (line 8) to destroy the contract and drain all the funds on the contract balance by invoking a built-in Solidity function `selfdestruct()` (line 9). As a result, all Ether remaining on the contract balance would be transferred to the attacker. This case demonstrates the serious consequences of bypassable modifiers in smart contracts.

Answer to RQ4: Analyzing the vulnerability results of SoMo enables us to identify three interesting security findings and perform a case study on a vulnerable contract of high value.

5 THREATS TO VALIDITY

In this section, we briefly discuss how to further improve SoMo.

Internal validity. As shown in §4.2, SoMo has false positives, which root from SoMo's taint analysis. We find that when encountering data structures like `array`, `mapping`, `struct`, or their composites, SoMo overly taints the involved elements and thus results in false positive findings. Additionally, circles exist on the constructed MDG, which may result in infinite loops during path slicing and consequently SoMo yields "timeout" when handling them. Though such infinite loops are rare, neglecting them leads to potential false negatives. To mitigate these threats, we plan to improve SoMo by incorporating finer-grained taint policy and by limiting the number of iteration steps when performing path slicing.

External validity. SoMo heavily relies on the analysis outputs of Slither; nevertheless, Slither fails to analyze a few contracts, which prevents SoMo from being applied and results in over 100 failure cases in our evaluation. To mitigate this threat, we plan to collaborate with the Slither team to enhance their tool. This will not only improve SoMo's applicability, but also benefit the Solidity community as a whole. Furthermore, SoMo currently struggles to process contracts that include multiple source-code files, because Slither lacks the capability to compile such contracts. To address this issue, we plan on extending Slither's functionality to support the compilation of multi-file contracts in the future.

6 RELATED WORK

Smart contract programs are known to be exploitable under various infamous vulnerabilities, such as the transaction order dependency (TOD) and reentrancy attacks [2, 40]. With this regard, numerous studies have been conducted to detect smart contract vulnerabilities via both static analysis [12, 22, 31, 40, 44, 48, 52, 58] and dynamic testing [17, 28, 30, 34, 43, 51] approaches. Sailfish [12] forms the so-called storage dependency graph (SDG) to enable static analysis of smart contract vulnerabilities. Clairvoyance [58] first conducts empirical study to summarize a comprehensive set of reentrancy patterns; it then performs cross-contract analysis to detect reentrancy bugs with high accuracy. As for dynamic testing-based approaches,

Reguard [34] focuses primarily on using fuzzing to detect reentrancy, while ContractFuzzer [30], ConFuzzius [51], and sfuzz [43] feature a wider range of bug patterns and higher vulnerability detectability. In addition to offline vulnerability detection, Sereum [46] prevents the DAO attack during runtime with runtime validation. Chen et al. [16] designs an EVM runtime monitoring framework with high applicability, and Solythesis [33] is designed with a focus on minimizing runtime validation overhead.

teEther [32], Ethainter [13], SPCon [37], and AChecker [25] are the most related studies. teEther formulates smart contract vulnerabilities in a general manner; it features automated detection and exploit generation on the bytecode level. However, teEther does not explicitly model contract modifiers. Ethainter and SPCon both take modifiers into account. Ethainter encodes information flow propagations as Datalog rules, whereas SPCon primarily relies on mining vulnerabilities from prior transactions. Another recent work, AChecker [25], was proposed concurrently with our study to statically detect access control vulnerabilities. Similar to SPCon, AChecker focuses on inferring role-related access control policies, such as checking the contract owner and address. That said, it only covers the role/address related modifiers, i.e., the U1 and U2 categories in Table 3. Moreover, AChecker relies on teEther [32] to analyze smart contracts at the bytecode level, which limits its access to source-code-level modifier information. In comparison, SoMo leverages the bypassable characteristic to accurately detect all types of insecure modifiers without the need of domain-specific knowledge (as in AChecker and SPCon) and fixed patterns (as in Ethainter and teEther).

VETSC [20] proposes a UI-driven method to extract DApps semantics and statically vets the inconsistency between UI and contract code. VRust [18] detects contract vulnerabilities on the Solana platform. Ghaleb et al. [24] present eTainter, a static analyzer to detect gas-related vulnerabilities in contract bytecode. Empirical efforts are spent to analyze the usage of revert statements in Solidity [36]. TXSpector [60] and SmartTest [49] are both vulnerable transaction sequence mining tools. NPChecker [53] models the nondeterminism in smart contracts that can result in payment bugs. SolType [50] presents a refinement type system for Solidity to prevent the arithmetic flows. SGUARD [42] applies runtime verification to automatically fix four kinds of vulnerabilities with minor overhead. Das et al. [19] conducted the first study on security issues in the NTF ecosystem, which heavily relying on the functionality of smart contracts. Recent studies also explored using AI-based approaches to detect vulnerability patterns [14, 29, 35, 38, 39, 41, 61].

7 CONCLUSION

In this paper, we conducted an empirical security analysis of custom function modifiers in real-world Ethereum smart contracts. To do that, we proposed a novel tool, SoMo, specialized for modifier analysis, which constructed modifier dependency graph (MDG) to cover all the modifier-related control/data flows, generated symbolic path constraints over MDG, and iteratively tested each candidate entry function. Our experiments showed that SoMo outperformed the state-of-the-art SPCon tool in detecting insecure modifiers, and successfully detected 411 vulnerable contracts with bypassable modifiers from 62,464 real-world contracts. The results also revealed three interesting security findings about the root causes of common modifier-related vulnerabilities. Furthermore, we conducted modifier-specific NLP analysis and clustering to identify nine types of major modifier usage in the wild. In the future, we plan to further enhance SoMo's analysis precision and robustness, as well as help developers mitigate modifier-related security risks.

ACKNOWLEDGEMENTS

We thank all the reviewers for their detailed and constructive comments. This work was partially supported by a direct grant (ref. no. 4055127) and a TDLEG grant (ref. no. 4170890) from The Chinese University of Hong Kong. The HKUST authors were supported in part by RGC RMGS under the contract RMGS22EG02.

REFERENCES

[1] 2014. Using a Divide and Conquer algorithm to Split Text without Spaces into List of Words. https://stackoverflow.com/a/21308255/197165/.
[2] 2016. The Dao Attack. https://www.coindesk.com/learn/understanding-the-dao-attack/.
[3] 2017. The Parity Wallet Bug. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7.
[4] 2023. Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/.
[5] 2023. EtherScan. https://etherscan.io/.
[6] 2023. sol-select. https://github.com/crytic/solc-select.
[7] 2023. The Solidity Document. https://docs.soliditylang.org/.
[8] 2023. Solidity Programming Language. https://soliditylang.org/.
[9] 2023. spaCy: Industrial-Strength Natural Language Processing. https://spacy.io/.
[10] 2023. WordNet by Princeton University. http://wordnet.princeton.edu/.
[11] 2023. Z3. https://github.com/Z3Prover/z3.
[12] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *Proc. IEEE Symposium on Security and Privacy*.
[13] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proc. ACM PLDI*.
[14] Jiachi Chen. 2020. Finding ethereum smart contracts security issues by comparing history versions. In *Proc. IEEE/ACM ASE*.
[15] Mengjie Chen, Daoyuan Wu, Xiao Yi, and Jianliang Xu. 2021. AGChain: A Blockchain-based Gateway for Permanent, Distributed, and Secure App Delegation from Existing Mobile App Markets. *CoRR arXiv* abs/2101.06454 (2021).
[16] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2017. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proc. ISOC NDSS*.
[17] 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 *Proc. IEEE/ACM ASE*.
[18] Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. 2022. VRust: Automated Vulnerability Detection for Solana Smart Contracts. In *Proc. ACM CCS*.
[19] Dipanjan Das, Priyanka Bose, Nicola Ruaro, Christopher Kruegel, and Giovanni Vigna. 2022. Understanding Security Issues in the NFT Ecosystem. In *Proc. ACM CCS*.
[20] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications. In *Proc. ACM CCS*.
[21] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proc. WETSEB*.
[22] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proc. USENIX Security*.
[23] Brendan J. Frey and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. *Science* (2007).
[24] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proc. ACM ISSTA*.
[25] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proc. ACM ICSE*.
[26] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. [n. d.]. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proc. ACM OOPSLA*.
[27] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. 2020. What are the actual flaws in important smart contracts (and how can we find them)?. In *Springer FC*.

[28] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proc. ACM CCS*.
[29] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Transactions on Information Forensics and Security* (2021).
[30] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proc. IEEE/ACM ASE*.
[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proc. ISOC NDSS*.
[32] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proc. USENIX Security*.
[33] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proc. ACM PLDI*.
[34] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proc. ACM ICSE*.
[35] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *Proc. IEEE/ACM ASE*.
[36] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *Proc. IEEE/ACM ASE*.
[37] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding Permission Bugs in Smart Contracts with Role Mining. In *Proc. ACM ISSTA*.
[38] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In *Proc. IJCAI*.
[39] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *CoRR* (2021).
[40] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proc. ACM CCS*.
[41] Hoang H Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2023. MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. In *Proc. ACM MSR*.
[42] Tai D Nguyen, Long H Pham, and Jun Sun. 2021. SGUARD: towards fixing vulnerable smart contracts automatically. In *Proc. IEEE Symposium on Security and Privacy*.
[43] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proc. ACM ICSE*.
[44] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proc. ACM ACSAC*.
[45] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *Proc. USENIX Security*.
[46] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proc. ISOC NDSS*.
[47] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proc. ACM POPL*.
[48] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proc. ACM CCS*.
[49] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution.. In *Proc. USENIX Security*.
[50] Bryan Tan, Benjamin Mariano, Shuvendu K Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. In *Proc. ACM POPL*.
[51] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *Proc. IEEE European Symposium on Security and Privacy*.
[52] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proc. ACM CCS*.
[53] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in ethereum smart contracts. In *Proc. ACM OOPSLA*.
[54] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. 2021. SoK: Decentralized Finance (DeFi). *CoRR arXiv* abs/2101.08778 (2021).
[55] Gavin Wood. 2023. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Yellow paper* (2023).
[56] Daoyuan Wu, Debin Gao, Rocky KC Chang, En He, Eric KT Cheng, and Robert H Deng. 2019. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *Proc. ISOC NDSS*.
[57] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. 2021. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid. In *Proc. IEEE DSN*.
[58] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2021. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proc. IEEE/ACM ASE*.
[59] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proc. IEEE Symposium on Security and Privacy*.
[60] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *USENIX Security*.
[61] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy Vulnerability Detection and Localization: A Deep Learning Based Two-phase Approach. In *Proc. IEEE/ACM ASE*.