# A Large-scale Empirical Study of Low-level Function Use in Ethereum Smart Contracts and Automated Replacement

by

Rui Xi

B.Eng., Sun Yat-sen University, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Applied Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

June 2022

© Rui Xi, 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**A Large-scale Empirical Study of Low-level Function Use in Ethereum Smart Contracts and Automated Replacement**

submitted by **Rui Xi** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

**Examining Committee:**

Karthik Pattabiraman, Professor, Electrical and Computer Engineering, UBC
*Supervisor*

Zehua Wang, Adjunct Professor, Electrical and Computer Engineering, UBC
*Supervisory Committee Member*

# Abstract

The Ethereum blockchain stores and executes complex logic via smart contracts written in Solidity, a high-level programming language. The Solidity language provides features to exercise fine-grained control over smart contracts, termed low-level functions. However, the high-volume of transactions and the improper use of low-level functions lead to security exploits with heavy financial losses. Consequently, the Solidity community has suggested secure alternatives to low-level functions.

In this thesis, we first perform an empirical study on the use of low-level functions in Ethereum smart contracts. We study a smart contract dataset consisting of over 2,100,000 real-world smart contracts. We find that low-level functions are widely used and that 95% of these uses are gratuitous, and are hence replaceable. We then propose GoHigh, a source-to-source transformation tool to eliminate low-level function-related vulnerabilities, by replacing low-level functions with secure high-level alternatives. Our experimental evaluation on the dataset shows that, among all the replaced contracts, about 80% of them do not introduce unintended side-effects, and the remaining 20% are not verifiable due to their external dependencies. Further, GoHigh saves more than 5% of the gas cost of the contract after replacement. Finally, GoHigh takes 7 seconds on average per contract.

# Lay Summary

Smart contracts are increasing in adoption in many blockchains. The programming language of smart contracts, Solidity, is known to include vulnerable features, including low-level functions, and can lead to great financial loss. In this thesis, we start with a large-scale empirical study of low-level function use in smart contracts and find that they are widely used and the majority of them are gratuitous. Thus the low-level functions can and should be replaced by Solidity guideline-recommended high-level alternatives. Finally, we propose a technique, GoHigh, to automate the replacing process. Our tool will aid in patching existing smart contracts which use low-level functions.

# Preface

This thesis is the result of work carried out by myself, with the guidance and mentorship of Dr. Karthik Pattabiraman. All chapters are based on the papers published below. I was responsible for conceiving the ideas, designing and conducting experiments, compiling the results, and writing the paper. My advisor was responsible for overseeing the project, providing feedback, and writing parts of the paper.

- Rui Xi and Karthik Pattabiraman, "When They Go Low: Automated Replacement of Low-level Functions in Ethereum Smart Contracts", *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022. (Acceptance Rate: 36%)*

- Rui Xi and Karthik Pattabiraman, "A Large-scale Empirical Study on Low-level Function use in Ethereum Smart Contracts and Automated Replacement", *In submission to a journal.*

# Table of Contents

**Abstract** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**

**Lay Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iv**

**Preface** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **v**

**Table of Contents** . . . . . . . . . . . . . . . . . . . . . . . . . . . **vi**

**List of Tables** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **ix**

**List of Figures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **x**

**List of Abbreviations** . . . . . . . . . . . . . . . . . . . . . . . . . **xiii**

**Acknowledgments** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xv**

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**
1.1 Smart Contract Vulnerability in Solidity . . . . . . . . . . . . . . 2
1.2 Existing Patching Techniques . . . . . . . . . . . . . . . . . . . . 3
1.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Contribution and Summary . . . . . . . . . . . . . . . . . . . . . 7

**2 Background and Related Work** . . . . . . . . . . . . . . . . . . . . **10**
2.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1.1 Blockchain . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1.2 Ethereum and Smart Contract . . . . . . . . . . . . . . . . 11

2.1.3 Solidity Language . . . . . . . . . . . . . . . . . . . . . . 12
2.1.4 Security Issues in Solidity Smart Contracts . . . . . . . . 13
2.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2.1 Empirical Study . . . . . . . . . . . . . . . . . . . . . . . 16
2.2.2 Detection . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.2.3 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.4 Patching . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2.5 Insecure Features in Other Languages . . . . . . . . . . . 21
2.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**3 Low-Level Functions: Definition and Empirical Study . . . . . . . . 23**
3.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.1.1 Selection and Definition of Low-level Functions . . . . . 23
3.1.2 Low-Level Functions Included . . . . . . . . . . . . . . . 27
3.2 Empirical Study . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.1 Dataset . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.2 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**4 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33**
4.1 Challenge and Contribution . . . . . . . . . . . . . . . . . . . 33
4.2 Abstract Syntax Tree (AST) Generation . . . . . . . . . . . . . 34
4.3 Pattern Matching . . . . . . . . . . . . . . . . . . . . . . . . 35
4.4 Replacement . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.6 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42**
5.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . 42
5.1.1 Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.1.2 Hardware and Software . . . . . . . . . . . . . . . . . . . 44
5.2 Research Questions (RQs) . . . . . . . . . . . . . . . . . . . 44

5.3 Results . . . 46
5.3.1 RQ1: Observations from Recent and Latest Datasets . . . 46
5.3.2 RQ2: Coverage of GoHigh's replacement . . . 47
5.3.3 RQ3: Compiler Warnings before and after Replacement . 49
5.3.4 RQ4: Unintended Side-Effects After Replacement . . . . 52
5.3.5 RQ5: Gas Cost Overhead of GoHigh . . . 55
5.3.6 RQ6: Performance of GoHigh . . . 57
5.4 Summary . . . 57

**6 Discussion . . . 59**
6.1 Possible Reasons Why Developers Violate the Guideline . . . 59
6.1.1 Gas Consumption Misunderstanding . . . 59
6.1.2 Overly Restrictive Guidelines . . . 61
6.1.3 Vulnerable Access Control Library . . . 62
6.1.4 Upgradeability and Proxy . . . 63
6.2 Threats to Validity . . . 64
6.2.1 External Threat . . . 64
6.2.2 Internal Threat . . . 65

**7 Conclusion and Future Work . . . 67**
7.1 Conclusion . . . 67
7.2 Future Work . . . 68
7.2.1 Inline Assembly (Inline Assembly (IA)) . . . 68
7.2.2 Upgradeable Proxy . . . 68
7.2.3 Access Control . . . 69

**Bibliography . . . 70**

# List of Tables


Table 1.1 The overall space of the contracts. . . . . . . . . . . . . . . . . 8

Table 3.1 Different categories of warnings in the Solidity documentation, and their characteristics for inclusion in our study. . . . . . . . 25

Table 4.1 The representations of Deprecated `send` (DS) and constant string Low-level `call` (LC). . . . . . . . . . . . . . . . . . . . . . . . 35

Table 4.2 The patterns of DS and LC for replacement, and the percentages of their occurrence in the base dataset. . . . . . . . . . . . . . 38

Table 5.1 The datasets used in the experiments. . . . . . . . . . . . . . . . 44

Table 5.2 The distribution of low-level function. . . . . . . . . . . . . . 46

Table 5.3 The distribution of complier warnings change before and after replacement. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Table 5.4 The success rate in replaying transactions to replaced contracts. 53

Table 5.5 The gas used overhead introduced by GoHigh. . . . . . . . . . 55

Table 6.1 The gas consumption of calling ERC-20 `transfer()` in three different methods. . . . . . . . . . . . . . . . . . . . . . . . . 60

# List of Figures


Figure 1.1 The monthly statistics of Ethereum transactions volume and market capitalization from March 2016 to December 2021. . . 2
Figure 1.2 The example of a vulnerable smart contract. . . . . . . . . . . 2
Figure 1.3 The example of a vulnerable smart contract patched by SmartShield. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
Figure 1.4 The example of a vulnerable smart contract patched by EVMPatch. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
Figure 1.5 The example of a vulnerable smart contract patched by our approach. The low-level function send() in line 6 is replaced by the high-level alternative, transfer(), as per the Solidity guidelines. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Figure 2.1 An example of Solidity smart contract. . . . . . . . . . . . . . 13
Figure 2.2 Examples on Solidity smart contract issues. . . . . . . . . . . 16

Figure 3.1 An example of a warning box in Solidity documentation. . . . 24
Figure 3.2 Examples of low-level functions. . . . . . . . . . . . . . . . . 28
Figure 3.3 Low-level functions usage in the base dataset in (a) all contracts, and (b) unique contracts . . . . . . . . . . . . . . . . . 30
Figure 3.4 Examples of two subcategories of LCs. . . . . . . . . . . . . . 30
Figure 3.5 The distribution of low-level functions in the base dataset in (a) all contracts, and (b) unique contracts. . . . . . . . . . . . 31

Figure 4.1 One replacement does not apply to every pattern. . . . . . . . 34
Figure 4.2 The structure of an AST of a Solidity smart contract. . . . . . 35

Figure 4.3 Difference between if-not and if-clause pattern. . . . . . . . . 37
Figure 4.4 The example before replacement. . . . . . . . . . . . . . . . 39
Figure 4.5 Example's AST (a) before replacement, and (b) after replacement. 40
Figure 4.6 The example after replacement. . . . . . . . . . . . . . . . . 41

Figure 5.1 Examples of (a) `Proxy` contract, (b) `Forwarder` contract and (c) `ForwardERC20` contract. . . . . . . . . . . . . . . . 48
Figure 5.2 Low-level functions usage in the **recent** dataset for (a) all contracts and (b) unique contracts. . . . . . . . . . . . . . . . . . 48
Figure 5.3 The distribution of low-level functions in the **recent** dataset for (a) all contracts and (b) unique contracts. . . . . . . . . . . . 48
Figure 5.4 Low-level functions usage in the **latest** dataset for (a) all contracts and (b) unique contracts. . . . . . . . . . . . . . . . . . 49
Figure 5.5 The distribution of low-level functions in the **latest** dataset for (a) all contracts and (b) unique contracts. . . . . . . . . . . . 49
Figure 5.6 Change in compiler warnings for the base dataset for (a) DS, and (b) Constant String LC. . . . . . . . . . . . . . . . . . . . 50
Figure 5.7 Warning Added after replacement by GoHigh. . . . . . . . . . 51
Figure 5.8 An example on special comment that disables low-level calls. 51
Figure 5.9 Change in compiler warnings for the recent dataset for (a) DS, (b) Constant String LC. . . . . . . . . . . . . . . . . . . . . . 52
Figure 5.10 Change in compiler warnings for the latest dataset for (a) DS, (b) Constant String LC. . . . . . . . . . . . . . . . . . . . . . 52
Figure 5.11 An example of a Solidity smart contract that contains external dependency. . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Figure 5.12 An example when Solidity compiler cannot estimate the actual gas used. . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Figure 5.13 The histogram of GoHigh's performance across smart contracts. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

Figure 6.1 An example of calling ERC-20 `transfer()` in three different methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
Figure 6.2 The different methods in transferring Ether. . . . . . . . . . . 61

Figure 6.3 An example of a contract reusing logics in a library contract. . 62

# List of Abbreviations

**ABI** Application Binary Interface

**AO** Arithmetic Operation

**AST** Abstract Syntax Tree

**BT** Block Timestamp

**CF** Cryptographic Function

**CFG** Control Flow Graph

**CS** Contract State

**DAO** Decentralized Autonomous Organization

**DOS** Denial of Service

**DS** Deprecated `send`

**EOA** Externally Owned Account

**ETH** Ether

**EVM** Ethereum Virtual Machine

**I1** Arithmetic Underflow and Overflow Issue

**I2** Reentrancy Issue

**I3** Unchecked Return Value

**I4** Use of Low-Level and Deprecated Solidity Functions

**I5** Block Gas Limit Issue

**IA** Inline Assembly

**IR** Intermediate Representation

**LC** Low-level `call`

**NLP** Natural Language Processing

**POW** Proof of Work

**USD** U.S. Dollar

**XML** Extensible Markup Language

# Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Karthik Pattabiraman for his constant support and invaluable guidance throughout my master's program. He has helped me refine research ideas and provided direction to build on top of them and execute to fruition. His dedication to high-quality research spurred me on pursuing challenges.

In addition to my advisor, I would like to thank my thesis examining committee, Dr. Julia Rubin and Dr. Zehua Wang, for their thought-provoking questions and valuable feedback on this thesis. I would like to thank Asem Ghaleb for his guidance in conducting experiments and other colleagues from the Dependable Systems Lab for their thoughtful discussions and constructive feedback.

Last but not least, I want to express my gratitude to my parents, whose unconditional support is the reason I can be here today.

# Chapter 1

# Introduction

The blockchain is a distributed ledger system that persists the historical transactions issued on it, which enables the tamper-proof value transfer worldwide. Two prominent examples of such systems are Bitcoin [58] and Ethereum [17]. Blockchain technology is increasingly adopted in many industrial sectors (e.g., IoT [70], supply chain [27]) and even governmental financial organizations [25]. In the near future, with the rise of global value transfer, it is predicted that blockchain as financial infrastructure will soon become a reality [22, 97].

Ethereum is one of the most popular blockchain systems today. Figure 1.1 shows the Ethereum transaction volume (red line) over five years. The transaction volume soared from less than 1 million per month in 2017 to more than 40 million per month recently. We observe a similar trend in the total market capitalization of Ethereum. In November 2021, the Ethereum blockchain reached its highest market capitalization (blue line), at 568 billion U.S. Dollar (USD) [2].

The most defining feature of Ethereum is smart contracts, which are programs that can be executed on the Ethereum blockchain to modify the Ethereum blockchain's states. As of the end of 2021, there are $48,084,708$ smart contracts on Ethereum [1], and some of these have attracted millions of transactions. Smart contracts are usually written using Solidity, a Turing-complete language, and compiled down to an executable format known as Ethereum Virtual Machine (EVM) bytecode before being deployed on the Ethereum blockchain.

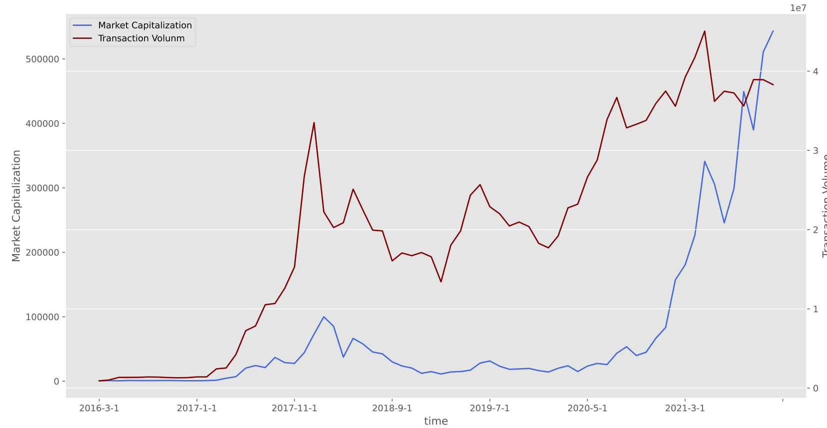


**Figure 1.1:** The monthly statistics of Ethereum transactions volume and market capitalization from March 2016 to December 2021.

∗ The unit of transactions volume is $10^7$ transactions (right y-axis). The market capitalization (left y-axis) is in USD.

## 1.1 Smart Contract Vulnerability in Solidity

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    receive() {
5      if (msg.value < claimPrice) revert();
6      king.send(calculateCompensation());
7      king = msg.sender;
8      claimPrice = msg.value;
9    }
10   function calculateCompensation() private {/*omitted*/}}
```

**Figure 1.2:** The example of a vulnerable smart contract.

In a Solidity smart contract, the main functionality is defined in `contract`, which is similar to classes in other object-oriented languages. An example of Solidity smart contract is given in Fig. 1.2, which is a simplified version of a real-world contract [10]. The contract is a game where competitors can claim the throne by transferring the most money (i.e., Ether (ETH), which is a virtual crypto-currency in Ethereum) to it. It contains two state variables (i.e., `king` and `claimPrice`) and

two functions (i.e., `receive()` and `calculateCompensation()`). Both the state variables are public, which means that they are publicly readable by other contracts and users. The `calculateCompensation()` function is private, which is not accessible to other contracts and users. The `receive()` is a special fallback function in Solidity: every time the contract receives ETH transfer, the `receive()` is invoked by default, and thus it is public without explicit declaration. This make possible for a contract to handle incoming ETH transfers.

Assuming that a user is claiming the throne by sending some ETH to the contract. After receiving an ETH transfer, the `receive()` is invoked automatically. In Line 6, the function first check whether the transferred ETH amount is greater than the highest price stored in the `claimPrice` variable. If the amount is less or equal to `claimPrice`, the execution will be reverted and the contract states will rollback to its initial state. Otherwise, in Line 6, the old king, whose address is stored in the `king` variable, gets his refund via the `send()` function. The refund amount is given by the `calculateCompensation()` function - we omit its implementation for simplicity concern. Finally, in Line 7 and 8, the new king is nominated and the new `claimPrice` is set to be the ETH transfer amount.

However, the example contract contains a missing "return value checks" vulnerability [4, 10]. The `send()` function does not throw an exception when the ETH transfer is not successful, instead, it returns a Boolean value `False` to represent the result. The inherited behavior of the `send()` function leads to a vulnerability: if the return value of the `send()` function is not checked, the execution of the `receive()` function will continue even if the ETH transfer fails. In the example, if the old king fails to receive the refund ETH transfer, the process of nominating a new king will continue, and hence the old king will lose his/her refund. This vulnerability leads to a real-world exploit in 2016, resulting in a loss of 42 ETH, which is about 168,000 USD.

## 1.2 Existing Patching Techniques

Smart contracts are attractive targets for attackers, as they deal directly with transacting ETH. Further, because smart contracts can be called by anyone who has the address of the contract, and the resulting transfer of ETH is automated, there are

few restrictions on the attackers. These factors have led to many attacks on smart contracts [56, 64], which have resulted in losses of millions of dollars.

The aforementioned exploit to the example contract succeeds because of the lack of proper checks on critical operations in smart contracts. Pattern-based detection approaches [28, 84], or symbolic execution approaches [53, 75] can be used for vulnerability detection. However, these approaches either fail to detect several instances of bugs [32], or they report high false-positive rates [37].

Patching techniques have proven effective in preventing possible exploits of smart contract vulnerabilities [74, 86]. Unlike detection, the patching technique patches smart contracts before their deployment, and thus prevents possible exploits from happening in the future. Such patching can be either manually implemented or automatically done. For a given smart contract, patching techniques aim to learn the vulnerable behavior and the patching strategies. Further, a patching engine can apply the strategies to the vulnerable smart contracts to mitigate the vulnerabilities. In this thesis, we aim to develop an automated patching tool to prevent possible exploits of smart contract vulnerabilities.

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    receive() {
5      if (msg.value < claimPrice) revert();
6      if(!king.send(calculateCompensation())) revert();
7      king = msg.sender;
8      claimPrice = msg.value;
9    }
10   function calculateCompensation() private {/*omitted*/}}
```

**Figure 1.3:** The example of a vulnerable smart contract patched by SmartShield.

To the best of our knowledge, there are only two techniques that have focused on patching techniques for smart contracts, namely SmartShield [86] and EVMPatch [74]. SmartShield patches the smart contracts by adding return value checks. For the example in Fig.1.3, SmartShield modifies Line 6 by adding an if-clause to check the return value of the refund transfer. If the transfer is not successful, the check introduced by SmartShield will revert the execution, and thus neutral-

```
contract KotET {
  address public king;
  uint public claimPrice = 100;
  receive() onlyOwner{
    if (msg.value < claimPrice) revert();
    king.send(calculateCompensation());
    king = msg.sender;
    claimPrice = msg.value;
  }
  function calculateCompensation() private {/*omitted*/}}
```

**Figure 1.4:** The example of a vulnerable smart contract patched by EVMPatch.

ize the vulnerability. On the other hand, EVMPatch patches the smart contracts by restricting access to vulnerable functions. For the example in Fig. 1.4, it adds an `onlyOwner` modifier to the `receive()` function. Hence, only the owner of the contract has access to the vulnerable `send()` function. The `onlyOwner` modifier is implemented by runtime checks on the caller's address.

Unfortunately, both of aforementioned patching techniques suffer from three critical shortcomings. First, SmartShield [86] requires manual inspection to locate the vulnerabilities, which requires human effort. Second, extra checks introduced by EVMPatch [74] lead to high gas overhead, which is the transaction fee in the Ethereum blockchain. Thirdly, both SmartShield and EVMPatch fail to eliminate the `send` function in the smart contract example, as they only add checks and access controls. Due to these shortcomings, these two patching techniques do not provide a scalable and efficient patching solution to the vulnerabilities.

## 1.3 Motivation

The vulnerable smart contracts usually originate from bad practices of Solidity developers [88], and a majority of these bad practices can be avoided by following the latest Solidity documentation and using the high-level features of Solidity [42]. Hence, we find that the root causes of smart contract vulnerabilities can be eliminated by following the Solidity official documentation. This insight motivates an alternative approach to patching smart contracts to follow the guidelines, rather than patching the vulnerabilities.

We extract a set of guidelines from the Solidity official documentation, which suggest using alternatives to replace the patterns that lead to insecure code. An example of such a pattern is a Solidity built-in feature, e.g., the `send` function, which is not recommended by the Solidity documentation because of known vulnerabilities in its use. We term those patterns *low-level functions* as all of these patterns include the use of low-level features of Solidity. The use of these features is no longer recommended since November 2018, when the Solidity version 0.5.0 was released. We provide a detailed selection criteria to extract the guidelines in Section 2.1. Although the low-level functions are not recommended by the guidelines, they are still supported by the EVM. Further, once a smart contract is deployed, it is very difficult to update or modify it, and hence the old, insecure patterns continue to proliferate. This is exacerbated by high levels of code reuse in smart contracts [21, 86].

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    receive() onlyOwner{
5      if (msg.value < claimPrice) revert();
6      king.transfer(calculateCompensation());
7      king = msg.sender;
8      claimPrice = msg.value;
9    }
10   function calculateCompensation() private {/*omitted*/}}
```

**Figure 1.5:** The example of a vulnerable smart contract patched by our approach. The low-level function send() in line 6 is replaced by the high-level alternative, transfer(), as per the Solidity guidelines.

The Solidity guidelines also recommend alternatives to the low-level functions, which are less vulnerable. We term them high-level alternatives. For example, the high-level alternative to the `send` function is the `transfer` function, which is free from the missing return value check vulnerability as the `transfer` function automatically throws an exception when the transfer fails. An example of the patching of the running example by our approach is shown in Fig. 1.5.

In summary, the ultimate goal of this thesis is to patch the vulnerable smart contracts by applying the guidelines defined in the Solidity official documentation to replace the low-level functions with their high-level alternatives.

## 1.4 Contribution and Summary

In this thesis, we characterize the use of insecure patterns that we term *low-level functions* in Solidity. We first perform a detailed analysis of the insecure code patterns that are mentioned in the Solidity documentation, and that have secure replacements (i.e., high-level alternatives). We find that these low-level functions are widely used in real-world smart contracts on the Ethereum blockchain. We further find that the majority of the uses of the low-level functions (more than 95%) are gratuitous for the smart contract's functionality, and can hence be replaced by high-level alternatives. However, there are many different patterns of such low-level functions, which make them challenging to replace in a uniform manner.

Based on the above insight, we then build an automated static replacement engine, GoHigh [1] that works on the Abstract Syntax Tree (AST) of the smart contract's source code to replace low-level functions with high-level alternatives. We distill the different patterns of low-level functions that are commonly used in smart contracts into AST templates, and propose replacements for the patterns taking into account their unique constraints, without requiring any programmer effort.

GoHigh covers multiple classes of security vulnerabilities in smart contracts detected by prior work [24, 53, 84]. None of these papers propose any solution to deal with the vulnerabilities however, and leave it up to the developer to rewrite the contracts. In comparison, GoHigh addresses both the return value check-related vulnerabilities [24, 53, 84] as well as the use of low-level and deprecated functions vulnerabilities [24], without requiring any effort from the developer. We categorize these vulnerabilities in different issues, and further elaborate on these vulnerabilities in Section 2.1.4.

*To the best of our knowledge, we are the first to empirically characterize the use of low-level functions in Ethereum smart contracts, and propose an automated technique to statically replace them with high-level alternatives.*

In summary, we make the following contributions in this thesis.

- We define a low-level function based on the guidelines in Solidity documentation, and empirically analyze the usage of low-level functions in a real-

[1] The name GoHigh are inspired by a quote from Michelle Obama in 2016, "When they go low, we go high".

world dataset consisting of nearly 150,000 Ethereum smart contracts ("base dataset"). We find that about 14% of the contracts in the base dataset use low-level functions, and that about 80% of these uses in the base dataset are gratuitous for the smart contract's functionality.

- We mine the base dataset to distill 11 distinct patterns of low-level functions that are used in the contracts, and represent the patterns as AST structures.
- We develop an automated, source-to-source transformation tool called GoHigh to detect low-level functions corresponding to the AST structures, and to replace the low-level functions with high-level alternatives[2].
- We analyze two more recent datasets of Ethereum smart contracts consisting of over 2,000,000 contracts ("recent dataset" and "latest dataset"), that were deployed well after the publication of Solidity guidelines to avoid low-level functions. We find that the number of smart contracts using low-level functions more than doubled (40%) compared to the base dataset (14%), and more than 95% of these uses were gratuitous for the contract's functionality, which covers 38% of the overall space of the contracts. We term this category as "replaceable" since all the contracts in this category have low-level functions that can be replaced with high-level alternatives. The overall space of the contracts is shown in Table 1.1.

**Table 1.1:** The overall space of the contracts.

| | Total | Use Low-level Functions | Replaceable | Verified | Not Verifiable |
|---|---|---|---|---|---|
| Percentage | 100% | 40% | 38% | 30.4% | 7.6% |
| Absolute Number | 2,150,000 | 860,000 | 817,000 | 653,600 | 163,400 |

- We find that the patterns distilled from the base dataset also applied to *all the contracts* in both the recent dataset and the latest dataset, and that they covered 100% of the contracts in both datasets. Further, GoHigh reduces the number of compiler warnings in the contracts by 4.9%. We also compare the behavior of the contracts after replacement by GoHigh by replaying

[2] Our artifacts are available at https://github.com/DependableSystemsLab/GoHigh

Ethereum transactions on the original and replaced contracts, and checking whether their externally visible states match. For the contracts that could be verified (about 80%), we find that all of the state changes matched after replacement by GoHigh in all datasets. The remaining 20% (which takes 7.6% of the overall space of the contracts) are not verifiable due to their external dependency on other contracts. Further, the gas costs of the contracts also reduced by 5% on average after the replacement performed by GoHigh. Finally, GoHigh takes between 0.1 and 60 seconds for replacement, with an average of 7 seconds per contract.

# Chapter 2

# Background and Related Work

In this chapter, we start by explaining the basic concepts of the blockchain and the smart contract. We follow up with security issues in the smart contract. We conclude with related work in the area of smart contract vulnerabilities.

## 2.1 Background

In this section, we first discuss the architecture and processes of blockchain, followed by an introduction to the main concepts of the Ethereum blockchain and Solidity language. Then, we provide an overview of security issues in smart contracts written using Solidity, namely Arithmetic Underflow and Overflow, Reentrancy, and Block Gas Limit.

### 2.1.1 Blockchain

At a high level, the blockchain is a ledger system based on a peer-to-peer network, which keeps transaction records to present the value transfer between accounts. Each peer, also known as a miner, in the blockchain takes the responsibility to validate and pack the transactions it receives, which compose a "block". Peers in the blockchain mine and append the latest block to previous blocks and broadcast it. If the latest block is valid and is accepted by the network, it will be a part of the ledger. Bitcoin [58] is the first implementation of blockchain, which employ Proof of Work (POW) as its consensus protocol. In POW protocol, all the peers should

race to find a nonce to generate a hash with a specific number of leading zero bits. The winner claims the privilege to build the block and receives the block reward.

### 2.1.2 Ethereum and Smart Contract

Ethereum is the second-most popular blockchain after Bitcoin, and its currency is ETH. The main difference between Bitcoin and Ethereum is that Ethereum supports smart contracts. A smart contract is a program running on the blockchain with a unique account address denoting smart-contract accounts. The user's account is called Externally Owned Account (EOA).

Transactions in Ethereum are classified into two categories, the ETH transfer transactions, and contract executions transactions, in terms of the recipient of the transactions. If a transaction is sent to an EOA, the transaction is an ETH transfer; otherwise, it is a contract execution. Ethereum charges a fee for all transactions as part of the incentive to the Ethereum blockchain peers, termed *gas*. For an ETH transfer, the gas usage is a flat 21,000 units (since 2015); while for contract executions, the gas cost depends on the storage and operation complexity of the target contracts.

The gas serves as the transaction fee for a blockchain transaction. The gas cost of a transaction $c$ is calculated by $c = u \times p$, where $u$ denotes the gas usage by transaction and $p$ denotes the gas price. The gas usage $u$ is determined by the operations of the executed smart contract function. For example, the `ADD` operation in Ethereum uses 1 unit of gas, while the `SLOAD` operation uses 200 units [18]. Note that the gas used by operations may change according to EVM versions. The gas price is a user-determined value of the user's willingness to pay for the transaction. The unit of gas price is Gwei ($10^{18}$ wei $= 10^9$ Gwei $= 1$ ETH).

The gas cost protects the Ethereum network from Denial of Service (DOS) attacks via sending a large number of transactions. To conduct a DOS attack on Ethereum, the attackers must pay a considerable amount of gas fee to let attack transactions be accepted. The gas cost also serves as one of the incentives to the Ethereum blockchain peers for executing smart contracts.

### 2.1.3 Solidity Language

Solidity is a programming language to write smart contracts. Solidity is a statically typed, object-oriented, and Turing-completed language with inheritance, library, and other features supported. Figure 2.1 shows an example of the structure of a Solidity smart contract. In Line 2, we define a contract called "Register", where a student can register his/her name. Before that, in Line 1, we first define the compiler version of Solidity. Because of the fast pace change of Solidity compiler versions, the latest compilers may not be backward-compatible with all smart contracts. In our case, the acceptable compiler versions are $>= 0.8.6 < 0.9.0$.

The Solidity language supports many built-in types, including `uint`, `bytes`, `string` and etc. Further, it also has special types such as `address` (in Line 3). The address is a unique global identifier for all EOAs and smart contracts. Any ETH transfers and contract executions must target a specific address. The "mapping" is a built-in key-value dictionary in Solidity. As shown in Line 4 and 5, there are two maps mapping from an `address` to a Boolean value and to a `string`, respectively. Solidity has access modifiers to control the accessibility to state variables and functions. The `public` modifier allows both internal and external access to variables and functions. However, the `private` modifier can only prevent access from other addresses inside the blockchain, but it cannot prevent users from reading the value from outside of the blockchain. Line 6 shows the header of a function. The function is an `external` function, which means that it can only be called from outside of the contract or through `this`, a reference to the current contract instance. The header also indicates that the function returns a Boolean value.

The function body is shown in Line 7-9. It contains a `require` statement, a value assignment and a `return` statement. Ethereum is a public blockchain that allows every user who has a connection to the Ethereum network to operate, and so users have access to all smart contract deployed on Ethereum by default. The `require` statement usually serves as an input sanitizer and an access control mechanism. If the value of the first parameter is not `true`, the `require` statement will revert the execution and will alert the user with the error message defined in the second parameter.

```
1  pragma solidity ^0.8.6;
2  contract Register {
3      address public admin;
4      mapping(address => bool) private registered;
5      mapping(address => string) public studentNames;
6      function registerName(string memory _name) external returns(
           bool){
7          require(!registered[msg.sender], "Register:: Already
               registered.");
8          studentNames[msg.sender] = _name;
9          return true;
10     }
11 }
```

**Figure 2.1:** An example of Solidity smart contract.

### 2.1.4 Security Issues in Solidity Smart Contracts

As programs are working on a permissionless system, i.e., by default all users have the capability to read, write and execute files in the system, Solidity smart contracts face many security issues [10]. Further, the high financial reward of exploiting the vulnerabilities makes smart contracts a ripe target for attackers. We discuss five typical security issues as follows. The first four issues (Arithmetic Underflow and Overflow Issue (I1) [3], Reentrancy Issue (I2) [5], Unchecked Return Value (I3) [4], and Use of Low-Level and Deprecated Solidity Functions (I4) [24]) are due to the nature of Solidity language; the last issue (DOS with Block Gas Limit Issue (I5) [6]) is due to the Ethereum blockchain and EVM design.

**Arithmetic Underflow and Overflow** (I1): An underflow/overflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For example, consider a number stored in an `uint8` type, i.e., in an 8-bit unsigned integer number ranging from 0 to $2^8 - 1$, if an arithmetic operation attempts to create a value $2^8$, which is outside of the range that an `uint8` type can represent, an overflow occurs. If the contract used the incorrect value in critical decisions, it will lead to financial loss. For example, assume a user withdraws 100 ETH from a smart contract bank, while her balance is only 99 ETH, which is stored in an `uint8` type. Without proper checks on the subtraction operation in calculating the balance, the value underflows to $2^8 - 1$ ETH. Further, due to the immutability of blockchain, the bank owner cannot correct the balance. One common solution to this issue is the SafeMath standard library [63].

**Reentrancy** (I2): The atomicity and sequentiality of a contract execution may be broken when a contract is interacting with an external contract. While developers may assume that a non-recursive function cannot be *re-entered* when it is revoked [10], this is not always the case because the fallback mechanism allows the callee contract to re-enter the caller function. This may result in unexpected behaviors. The fallback mechanism in Solidity smart contract is designed to handle ETH received, and function calls that do not match any defined functions. The fallback functions are usually used to redirect function calls and handle received ETH. There are two types of fallback function, `receive()` and `fallback()`.

The `ReentranceVictim` contract and the `ReentranceAttacker` contract in Fig.2.2 show an example of a pair of reentrancy victim and attacker. The `ReentranceVictim` contract is a bank that allows its users to withdraw ETH from. The `withdraw()` function first retrieves the ETH amount of deposit (Line 4). Then, it transfers the ETH to the address who calls the `withdraw()` function (Line 5). Finally, it resets the deposit amount to zero (Line 6). If the `withdraw()` function is executed atomically, the function is free from attack. However, because ETH transfer (Line 5) triggers the fallback mechanism in `ReentranceAttacker` (Line 11), resetting amount (Line 6) is not executed right after ETH transfer (Line 5). Instead, as the fallback function defined in contract `ReentranceAttacker` (Line 3), it re-enters the `withdraw()` function, and transfer the same amount of ETH for a second time. Noteworthy, the "Decentralized Autonomous Organization (DAO) hack" exploited this vulnerability in June 2016, and the "Agave hack" exploited the very same vulnerability in March 2022. Both of them led to millions of USD-worth ETH loss.

**Unchecked Return Value** (I3): As shown in Chapter 1, the lack of return value check will resume the function execution even if the `send` function is unsuccessful. If the ETH transfer accidentally fails or malicious users force it to fail, this vulnerability may cause unexpected behavior in the subsequent program logic. In the example we presented in Chapter 1, the consequence is that the old king fails to get his refund. Other than the `send` function, the low-level `call` functions have the same vulnerability. We further elaborate on this issue by giving code examples in Chapter 3.

**Use of Low-Level and Deprecated Solidity Functions** (I4): Using low-level

and deprecated functions can violate the specification in unexpected ways that effectively disable built-in protection mechanisms and introduce exploitable inconsistencies. In the Solidity official documentation, the low-level call and inline assembly are marked as the low-level functions and should be avoided whenever it is possible. Meanwhile, the `send` function shown in Fig. 1.2 is labeled as deprecated, and thus should not be used. The use of the low-level and deprecated functions introduces other issues (e.g., I3). Therefore, it should be avoided in developing Solidity smart contracts.

**Block Gas Limit** (I5): The Ethereum blockchain specifies a gas limit for each block. Recall that the block is a set of transactions that are packed and verified by the Ethereum peers. The sum of the the gas used of all transactions in one block must be lower than the gas limit, otherwise the block is not valid and will not be accepted by the Ethereum network. The block gas limit also constrains the maximum gas used of a single transaction. The gas used for iterating over a large array grows with the array size. If the gas used of querying the array exceeds the block gas limit, the query will never succeed.

The `BlockGasLimit` contract in Fig.2.2(c) shows an example of a coding pattern that may reach the block gas limit. The field `players` is a dynamic array to store all the players of the contract. Players add themselves to the contract via calling the `addPlayer()` function. The `claimWinner()` function will send the winner all the ETH stored in the contract. The winner is chosen as the player who has the largest address. When the contract is determining the winner, it loops over the array. If the array is so long that the iterating it drains all the gas, the execution will be reverted, and the winner can never claim his/her reward.

## 2.2 Related Work

In this section, we discuss related work on empirical analysis, vulnerability detection techniques, testing techniques for Solidity smart contracts, and dynamic features in other programming languages. We classify the related work into five categories (1) Empirical Study, (2) Detection, (3) Testing, (4) Patching, and (5) Insecure Features in Other Languages. First, we discuss the vulnerable features in Solidity smart contracts, followed by the detection techniques and their limitations.

```
1 contract ReentranceVictim {
2     mapping(address => uint) balance;
3     function withdraw() {
4         uint amount = balance[msg.sender];
5         payable(msg.sender).call{value: amount}("");
6         balance[msg.sender] = 0;}}
```

**(a)** An example of vulnerable contract with the reentrancy issue.

```
1 contract ReentranceAttacker {
2     function attack() {reentranceVictim.withdraw();} // start
        attack
3     receive() {reentranceVictim.withdraw();}}
```

**(b)** An example of attacker contract that exploits the reentrancy issue.

```
1 contract BlockGasLimit {
2     address[] players;
3     function addPlayer() {players.push(msg.sender);}
4     function claimWinner() {
5         address winner = address(0);
6         for (uint i=0; i<players.length; i++) if (players[i]>
            winner) winner=players[i];
7         payable(winner).transfer(this.balance); }}
```

**(c)** An example of vulnerable contract with the block gas limit issue.

**Figure 2.2:** Examples on Solidity smart contract issues.

Then, we discuss the existing testing and patching tools. Finally, we discuss the insecure features in other programming languages.

### 2.2.1 Empirical Study

Recent work carried on multiple empirical studies on Solidity programming language features and vulnerabilities. Atzei *et al* [10] categorized known vulnerabilities in Solidity smart contracts, which are caused by Solidity language features. Hwang and Ryu [42] studied smart contracts that use older version Solidity compilers. Their work reveals that more than 98% of real-world Solidity contracts use older version compliers without vulnerability patches. We also discuss vulnerabilities from deprecated functions and specifically focus on the low-level function in this thesis. Wang *et al* [88] summarized 41 common features used in real-world Solidity smart contracts. Their results indicate that high-level function invocations have been one of the most popular features in Solidity. And the low-level calls and send are used in 14.77% and 5.62% among all open-source Solidity smart

contracts, respectively, which complies with our observations.

### 2.2.2 Detection

There has been significant work in detecting vulnerabilities in smart contracts.

**Pattern-based Method**. Securify [84] uses both compliance and violation patterns to detect possible violations of the secure patterns in smart contracts. It provides four default sets of patterns, and a domain-specific language for user-defined patterns. SmartCheck [80] extracts syntax and semantic information from the smart contract's source code. It then utilizes XPath, a query language in searching nodes in Extensible Markup Language (XML) documents, to match vulnerable patterns. However, due to the increase of complexity in contract logic, SmartCheck does not work correctly for Solidity versions starting with 0.6.0 and is announced deprecated in 2020. MadMax [35] detects gas-related vulnerabilities (I5) from the Control Flow Graph (CFG), memory layout, and data structure of the smart contract. It implements a decompiler to transform EVM bytecode into Intermediate Representation (IR), and then employ IR rules to filter possible violation of gas-safe patterns. eTainter [33] employs taint analysis in gas-related vulnerability (I5) detection, with the consideration of protective patterns. Slither [28] is a static analysis framework that converts Solidity smart contract into an intermediate representation called SlithIR. Slither then uses data-flow analysis to detect bug patterns scoped within a function. The shortcoming is that Slither focuses on data-flow at function level and fails to do path reasoning, which leads to false alarms. SAILFISH [16] and SolType [79] detect state inconsistency and arithmetic overflow (I1), respectively, on top of SlithIR. Beillahi *et al* [13] propose an automated detection and patching tool on transaction order dependency vulnerabilities based on SlithIR. Clairvoyance [94] extends Slither via utilizing cross-function and cross-contract information, reaching both high accuracy and low false positive in detecting reentrancy bugs (I2).

**Symbolic Execution Method**. Oyente [53] is the first tool for detecting smart contract vulnerabilities using symbolic execution. It is capable to detect four types of vulnerabilities, i.e., mishandled exceptions, transaction-ordering dependence, timestamp dependence, and reentrancy (I2). Chen *et al* [20] extended Oyente by

combining with CFG construction and stack event analysis. The proposed DEFECTCHECKER detects 20 types of smart contract vulnerabilities. Se *et al* [75] combined symbolic execution with a language model to solve the path accessibility problem. However, their tool fails to detect reentrancy defects (I2), which remains one of the most severe vulnerabilities in smart contracts. RA [23] is a symbolic execution-based analyzer detecting only reentrancy vulnerability. The authors developed an emulator, which emulates inter-contract reentrancy attacks (I2), which is the first work that can detect inter-contract attacks without executing the smart contracts. ETHBMC [30] provides a bounded model checker based on symbolic execution to automatically generate exploits to inter-contract vulnerabilities. The authors showed ETHBMC's effectiveness by running experiments on both toy examples and real-world smart contracts. The latter experiment successfully extracted ETH from more than 5,000 real-world smart contracts. Similarly, Feng *et al* [29] presented the state dependency of exploits as summaries and employed symbolic execution to query possible exploits in Solidity smart contracts.

**Machine Learning-based Method**. Extensive work has explored the possibility of leveraging machine learning techniques in detecting Solidity smart contract vulnerabilities, including K-nearest neighbors algorithm [93], random forest [92] and decision tree [87]. In terms of neural network, Liu *et al* [50, 51] first employed Graph Neural Network to extract features from the CFG and the data-flow semantics of the source code of smart contracts. However, the proposed tools work on the function level, which means that they fail to detect any inter-function and inter-contract vulnerabilities. Peculiar [89] targets the crucial data flow graph of Solidity smart contacts, and involves a pre-trained sequence-to-sequence tagging neural network model to process both the source code and data flow graph. Other attempts on transferring Natural Language Processing (NLP) knowledge into the smart contract vulnerability detection also considered different network architectures, e.g., word2vec [9, 38], LSTM [76], and BERT [43].

**Evaluation of Detection Tool**. The idea of "analyzing programming analyzers" [19] is attractive since most tools do not produce "the right answer" all the time because of over-fitting, algorithmic limitation, and engineering trade-offs. Ghaleb and Pattabiraman [32] evaluated a wide range of Solidity smart contract analysis tools [24, 28, 53, 80, 84] using bug injection, revealing that the evaluated tools fail

to detect several instances of bugs despite the claims made by the papers describing the tools. Meanwhile, all the evaluated tools report many false positives. Durieux *et al* [26] evaluated nine automated analysis tools [24, 28, 53, 57, 61, 80–82, 84] using an annotated dataset, and a real-word contract dataset. Their results showed that the existing tools are only able to detect together 42% of the vulnerabilities from the annotated dataset while reporting low agreement among the evaluated tools. Yu *et al* [96] specifically analyzed the false positives on reentrancy reported by [28, 53], and derived five path filters to eliminate false positives. Groce *et al* [37] compared three Solidity smart contract analysis tools [28, 80, 84] by producing mutants of real code. Their experiments also revealed that Solidity analysis tools report high false-positive rates. However, their experiments are based on 100 randomly selected smart contracts from EtherScan, which may not be representative.

### 2.2.3 Testing

**Fuzz Testing**. Fuzzing is an testing technique used to test programs with randomly generated input data. Reguard [49] detects reentrancy vulnerability (I2) using fuzzing. Reguard translates Solidity smart contracts into C++ files, and then uses fuzzing engines (e.g., AFL and LibFuzzer) to generate exploits. However, the authors evaluated Reguard using only five Ethereum contracts, which is insufficient to reflect the effectiveness of the fuzzing technique in testing smart contracts. Jiang *et al* [45] proposed a fuzzing framework specific to Solidity smart contracts, detecting 7 categories of vulnerabilities including reentrancy (I2). However, the proposed tool, ContractFuzzer, is only able to reenter the same function as the initial call, which greatly narrows down the search space for detecting reentrancy vulnerability. ReDefender [65] extends ContractFuzzer by leveraging the storage state and depth of the call stack to automatically generate test cases with high accuracy in detecting reentrancy vulnerabilities (I2). Nguyen *et al* [59] proposed a feedback-based adaptive fuzzer called sFuzz to improve the branch coverage efficiency. Ji *et al* [44] extended sFuzz using dynamic taint analysis, which increases the covered branches by 6%. Echidna [36] and Foundry [66] are fuzzing frameworks widely used in industry, both of which feature efficient test generation with

simple configuration and user-friendly manuals.

**Formal Verification**. Formal verifications are based on formal methods of mathematics and are able to provide formal proof of the correctness of the software program with respect to its specification. Bhargavan *et al* [15] proposed the first functional programming language, named F* to analyze and verify both the runtime safety and the functional correctness of Solidity smart contracts. FSolidM [54] is a formal verification tool on Solidity smart contracts that allows its user to design a smart contract as a finite state machine and then transform it into Solidity. VeriSolid [55] extends FSolidM by introducing the aspect of formal verification into the tool, which provides the user with the ability to specify intended behavior in the form of liveness, deadlock freedom, and safety properties. However, both FSolidM and VeriSolid restrict users with limits of the provided templates and they only verify the state machine, instead of the smart contract's properties. VerX [69] also extends FSolidM in verifying temporal properties in the finite state machine. Park *et al* [67] proposed KEVM, a complete formal semantics of the EVM powered by K-framework, to reason about all possible corner-case behaviors of the EVM bytecode. Other than K-framework, existing work also employed Isabelle/HOL [71], Coq [77] and SMT solvers [7] to formally prove the correctness of Solidity smart contracts. Solythesis [47] is a source to source runtime validation tool on Solidity smart contracts. However, it requires an EVM update, and all Ethereum peers must agree to migrate to the updated version in order for the protection to take effect.

### 2.2.4 Patching

As we already introduced two major patching techniques, SmartShield and EVMPatch, in Chapter 1, we recapitulate them quickly and focus more on their extensions in this section. SmartShield [98] is the first tool on protecting low-level calls by adding a return value check to each of them. Elysium [83] extends SmartShield by adding two patching templates on access control and denial of service, but fails to address the shortcomings of SmartShield. Another recent technique, EVMPatch [74] protects not only the low-level calls statement but the whole function by adding access control to it. The access controls check whether a contract is

properly initialized before being used (whether the owner os correctly configured), and restrict the caller of sensitive and vulnerable functions (e.g., initialize, self-destroy, functions that contain low-level functions). sGuard [60] and HCC [34] extend EVMPatch by extracting a CFG and data-flow pattern to patch reentrancy and arithmetic overflow in the bytecode level.

### 2.2.5 Insecure Features in Other Languages

Insecure features in other programming languages have been well-studied. For example, Richards *et al* performed a large-scale study on the usage of `eval` in JavaScript [73]. Prior to that work, they carried out a more general analysis on more dynamic features of JavaScript [72]. The low-level functions in Solidity are similar to the `eval` call in JavaScript, which allows developers to convert strings into executable code. Similarly, Holkner and Harland [41] found that dynamic code execution is also widely adopted in Python. Livshits *et al* [52] conducted static analysis on Java reflection, including reflective calls whose functionality is similar to Solidity Low-level `call` (LC). Insecure calls in C code are frequent as well, e.g., Austin *et al*proposed a source-to-source code transformation tool to address the point and array access errors, and LibSafePlus [11] provides runtime protection against buffer overflows. However, low-level functions in Solidity are fundamentally different.

## 2.3 Summary

The native support of value transfer in Ethereum blockchain and the customizable Solidity smart contracts make them one of the best candidates for financial infrastructure. However, the emerging issues of Solidity smart contracts reflect a huge gap between the security and dependability expectation of financial infrastructure and the reality of the imperfect Ethereum blockchain and Solidity language. To address these issues, researchers carried out extensive studies in testing smart contracts and in detecting their vulnerabilities.

To date, most of the previous work in vulnerability detection against Solidity smart contract has been confined to detecting potential attacks [28, 35, 53, 80, 84, 89, 94], instead of providing patching. Furthermore, the detection techniques

report many false positives [26, 32, 37]. Existing work also shows that using high-level features in Solidity [88, 90] and following the documentation of the latest Solidity compiler with high-level features supported [42] help secure smart contracts. However, to the best of our knowledge, there are no tools for automated enforcing the documentation recommended features at the source code level. We aim to fill this gap with our GoHigh which enables the replacement of low-level functions with their high-level and documentation recommended alternatives, thus providing the capability to strengthen the existing smart contracts. In Chapter 3, we present the definition of low-level functions, and the analysis of their usage based on a smart contracts dataset.

# Chapter 3

# Low-Level Functions: Definition and Empirical Study

In this chapter, we start by presenting the definition and selection criteria of *low-level functions*. We follow up with the empirical study on the use of them. We conclude with the key observations we derived from the empirical study.

## 3.1 Definition

In this section, we first define *low-level functions*, and present the criteria of our selection of the same by distilling the Solidity documentation. We then delve into the three categories of low-level functions considered in this thesis, namely Deprecated `send`, Low-level `call` and Inline Assembly. Finally, we outline potential vulnerabilities of low-level functions with examples.

### 3.1.1 Selection and Definition of Low-level Functions

We define a low-level function as one that has a warning against its use in the Solidity documentation, and can be detected and replaced by syntax-level analysis. To extract low-level functions, we systematically study the official Solidity documentation[1] and collect all the alerts highlighted in warning boxes in the *Language Description* section of the documentation (the other sections have to do with tu-

[1] https://docs.soliditylang.org/en/v0.8.6

**Warning**

Before version `0.5.0` a right shift `x >> y` for negative `x` was equivalent to the mathematical expression `x / 2**y` rounded towards zero, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

**Figure 3.1:** An example of a warning box in Solidity documentation.

torials for beginners and implementation details of the EVM, neither of which are relevant for us).

We find that there are 28 warning boxes in total in the documentation[1]. Of the 28 warnings, we observe that seven warnings (W2-3, W7, W16, W19, W24-25) are constrained to a specific range of compiler versions. We exclude such warnings as they can be eliminated with later versions of Solidity compilers. For example, in Figure 3.1, the **W2** says "Before version 0.5.0, a right shift ...", which means that it is confined to Solidity compilers version 0.4.x or earlier.

[1]We assign numbers to the warnings in the order of their appearance.

**Table 3.1:** Different categories of warnings in the Solidity documentation, and their characteristics for inclusion in our study.

| Name | Included | Vulnerabilities | Warnings | Criteria for Inclusion | | |
|---|---|---|---|---|---|---|
| | | | | No Mature Solutions | Real-world Exploits | No Semantic Knowledge |
| AO | ✗ | Underflow/overflow | W1, W4, W17 | ✗ | ✓ | ✓ |
| DS | ✓ | Mishandled exception<br>Existent check<br>Out-of-gas | W5, W10, W21 | ✓ | ✓ | ✓ |
| LC | ✓ | Mishandled exception<br>Reentrancy<br>Existent check<br>Argument Invalid | W6, W11-13, W18 | ✓ | ✓ | ✓ |
| IA | ✓ | Same as LC's | W26-28 | ✓ | ✓ | ✓ |
| CF | ✗ | Not unique signature | W8 | ✓ | ✗ | ✓ |
| BT | ✗ | Bad randomness | W14 | ✓ | ✓ | ✗ |
| CS | ✗ | Balance mismatch<br>Hidden state exposure<br>Unexpected copy behavior | W15, W20, W22-23 | ✓ | ✓ | ✗ |

We then group the remaining 21 warnings into seven categories, i.e., Arithmetic Operation (AO), Deprecated `send` (DS), LC, Inline Assembly (IA), Cryptographic Function (CF), Block Timestamp (BT) and Contract State (CS). Table 3.1 shows these categories. These categories are based on the corresponding operations. For example, warnings about the `send` function fall into the DS category, and warnings about arithmetic operations (e.g., add, subtract, shift and etc.) fall into the AO category (I1). The LC category includes warnings about low-level call family, while the IA category contains all warnings from the Inline Assembly subsection. The CF category and BT category are about the misuse of the `block.timestamp` attribute and the `ecrecover` cryptographic function, respectively. The CS category includes operations that influence the contract states, such as the contract's balance, state visibility, and self-destruct function.

We follow three criteria to select the low-level functions to focus on in this work. First, there should be no widely available, mature solutions to the problem, e.g., a practical and widely-used library. Second, the category should have real-world exploits. Some vulnerabilities are in functions that are seldom used by developers, and thus have no real-world exploits. Third, the problem must be addressable with simple syntactic replacement, and not require semantic knowledge of the contract. This is because, as a static analysis technique, our technique has no information about the contract's semantics beyond the source code.

Based on the above criteria, we choose three categories of warnings as low-level functions, namely DS, LC, IA. DS and LC both operate on `address` in Solidity, and share many similarities in their vulnerabilities, i.e., missing return values and existence check (I3). The documentation also suggests alternatives (e.g., transfer for send, high-level calls that operate on contract instances for low-level calls) to replace them. There have also been real-world vulnerabilities in both of these categories. For example, the DAO attack[2] exploited the reentrancy vulnerability (I2) in a low-level call and resulted in a hard-fork in Ethereum. Therefore, we include the DS and LC categories. We also include the IA category, as there is also a call instruction in inline assembly which is similar to the LC category, and thus it shares the same vulnerabilities as LC. Although there are no real-world exploits in

[2]https://www.coindesk.com/understanding-dao-hack-journalists

the IA category, many attacks in the DS and LC categories also apply to it.

We briefly describe why we exclude the other categories. AO (I1) has been extensively studied in the research community [8, 46], and can be easily mitigated by using the arithmetic functions in the Solidity math library named SafeMath [39] or by upgrading to Solidity version 0.8.x. There has also been extensive work on bytecode level rewriting [12, 31] to use such techniques. BT requires understandings of a contract's semantics for its mitigation. A block timestamp is a number filled by the block miner, and can be modified during block generation, which makes it vulnerable to malicious miners. While it is recommended to avoid using BT as a source of randomness, the specifics of when such use is acceptable depends on the contract's semantics. Similarly, CS require semantic knowledge for mitigation to determine whether a state of the contract is hidden. Therefore, we exclude these categories. Finally, we exclude CF as we could not find any real-world exploits in this category (at the time of writing).

In the following subsection, we describe the three categories of warnings in our definition of low-level functions in this thesis: DS, LC, and IA. *In the rest of this thesis, we refer to the warnings to avoid low-level functions as Solidity guidelines.*

### 3.1.2 Low-Level Functions Included

**Deprecated `send`.** Deprecated `send` is a member function of an address object that is used to send ETH to an address. Address is a built-in data type of Solidity, which supports a series of functions that directly interact with this address. By invoking DS, the contract sends ETH to the targeted Ethereum account (`address receiver` in Fig.3.2a). However, a failed transaction via DS does not trigger a rollback. Instead, it returns `false`, so the execution continues even though the transaction fails (I3). As a result, from the Solidity version 0.4.10 onwards, the documentation recommends that developers use a safer replacement for DS (e.g., `transfer`).

**Low-level `call`.** Low-level `call` is a member function of an address to call a function in this address (if any). It is used to call an arbitrary function, similar to a function pointer in C and C++. It is primarily used to interact with contracts that do not adhere to the Application Binary Interface (ABI), or to get more direct control

over the encoding. There are three functions for performing such LCs, namely `call`, `staticcall` and `delegatecall`. Figure 3.2b shows an example of an LC: it takes a single byte memory parameter (payload in the example) and returns the success condition as well as the data (I3). We focus on the `call` function, as it is used by 98.7% of contracts using the call functions in our smart contract dataset (Section 3.2).

**Inline Assembly.** Solidity allows developers to have fine-grained control on the memory allocation in contracts via IA, which is close to the EVM. Developers leverage this feature to save gas cost, especially for frequently-used contracts. IA also provides a `call` instruction. Figure 3.2c shows an example of a call in IA - it calls a contract using a hand-crafted payload (`calldata`) at a given address (`childContract`), returning 0 on error (e.g., out of gas) and 1 on success(I3)[3]. Crafting the payload may result in corrupted data if the encoding is not correctly implemented. Moreover, the call instruction can also be used to invoke arbitrary function calls by the smart contract, which lead to the same issues as LCs.

```
1 address payable receiver = address(0x123);
2 receiver.send(10);
```

**(a)** An example of DS.

```
1 address nameRegister = address(0x123);
2 bytes memory payload = abi.encodeWithSignature("register(string)"
    , "John");
3 nameRegister.call(payload);
```

**(b)** An example of LC.

```
1 //0x095ea7b3 == "approve(address,uint256)"
2 bytes memory calldata = abi.encodeWithSelector(0x095ea7b3, this,
    _childTokenId);
3 assembly {
4     let success := call(gas, childContract, 0, add(calldata, 0x20
        ), mload(calldata), calldata, 0)
5 }
```

**(c)** An example of IA.

**Figure 3.2:** Examples of low-level functions.

[3]https://docs.soliditylang.org/en/v0.8.6/yul.html#yul-call-return-area

## 3.2 Empirical Study

In this section, we perform an empirical study of smart contracts deployed on the Ethereum blockchain. We first describe the Solidity smart contract dataset and the underlying tools that we used to collect the data. Then, we study the frequency and the usage of low-level functions utilizing the dataset, followed by the results. Finally, we present two key observations that we derive from the results on the low-level functions.

### 3.2.1 Dataset

We use a real-world smart contracts published to the Ethereum blockchain, from March 1, 2016 to September 30, 2019[4]. Of the 16 million contracts in this dataset, $149,363$ contracts have their source code available - these constitute our dataset.

All of the contracts specify Solidity compiler versions from 0.4.0 to 0.5.4. The version information for a contract is important, as the contract will not compile if the compiler version does not match the declared one (in some cases, no compiler version is specified, and so we try compiling it with different versions to see if there is a match). Note that the guidelines in the documentation were introduced for Solidity compiler 0.4.0 and evolved from 2016 to 2018, before being finalized in November 2018 with the release of Solidity 0.5.0. Thus, these contracts were developed more or less simultaneously with the guidelines in the documentation.

We determine whether a contract is unique based on its MD5 checksum, as it has been shown that many contracts in Ethereum are duplicated [21, 86]. We find that there are $61,444$ unique contracts in our dataset (41% of the dataset). Therefore, we present results for both the unique and overall contracts.

### 3.2.2 Tools

We quantify the usage of low-level functions by collecting smart contract addresses using Ethereum-ETL [1] and performing offline analysis subsequently. The source code of the contracts is collected from Etherscan [2], a public platform that monitors every transaction recorded in Ethereum, including the creation of smart contracts and users' interaction with them.

[4] We use more recent datasets to evaluate GoHigh in Chapter 5.

### 3.2.3 Results

We start by studying the frequency of low-level functions in the smart contracts in the dataset. Shown in Fig.3.3a, *We find that among* 149,363 *contracts,* 13.8% *use low-level functions.* The percentage increases slightly to 17.2% if we consider the set of unique contracts, as shown in Fig.3.3b. These figures show that low-level functions are widely used in smart contracts.

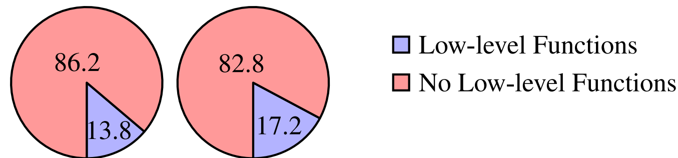


**Figure 3.3:** Low-level functions usage in the base dataset in (a) all contracts, and (b) unique contracts

We further perform detailed analysis of low-level functions. Before doing so, we subdivide LC into two sub-categories: (1) constant string LC and (2) arbitrary LC. The former refers to cases where the developer hard-codes the function to be called into a constant string, and invokes the function using an LC. The latter refers to cases where the call is made to an arbitrary function whose address is difficult to determine at compile-time. Figures 3.4 shows examples of the constant string LC and the arbitrary LC.

```
1 function constant_string_call() public{
2     address hardcode_address = address(0x123);
3     bytes memory hardcode_payload = abi.encodeWithSignature(
4         "register(string)", "John");
5     hardcode_address.call(hardcode_payload);
6 }
```

**(a)** An example of constant string LC.

```
1 function arbitrary_call(address arbitrary_address, bytes memory
    arbitrary_payload) public{
2     arbitrary_address.call(arbitrary_payload);
3 }
```

**(b)** An example of arbitrary LC.

**Figure 3.4:** Examples of two subcategories of LCs.

We calculate a more detailed usage statistics of low-level functions in Fig.3.5,

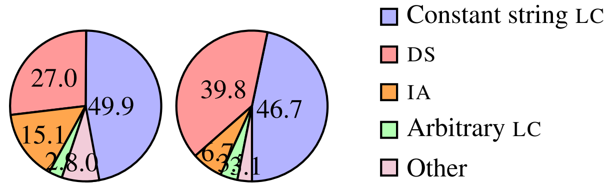


**Figure 3.5:** The distribution of low-level functions in the base dataset in (a) all contracts, and (b) unique contracts.

including the two sub-categories of LCs. We only consider the contracts that use at least one low-level function in this figure. As shown in Fig.3.5a, the constant string LC is used by the dominant fraction (49.9%) of all the contracts that use low-level functions. The DS is in second place at 27.0% and the IA is at 15.1%. "Other" in this figure stands for the contracts that include more than one low-level function categories. Note that the arbitrary LC is only found in 2.9% of the dataset. If projected to the entire dataset of smart contracts on Ethereum, the percentage of contracts that use arbitrary LC is only 0.40% (2.9% $\times$ 13.8%).

Figure 3.5b shows the detailed breakdown of low-level function usage among the unique contracts. We find that the percentage of DS increases from 27.0% to 39.8% in this dataset, while the percentage of IA halves, representing 6.7%. The percentage of arbitrary LC slightly increases from 2.9% to 3.7%, but it is still the least prevalent among the four categories, as was the case with the overall contracts.

## 3.3 Summary

We can make two key observations from the results.

1. Low-level functions are widely used in real contracts.

2. The majority of low-level functions are gratuitous and can be replaced by high-level ones.

The first observation follows directly from the results on the prevalence of low-level functions in both the overall contract set as well as the set of unique contracts on Ethereum that have their source code available. The second observation follows from the distribution of the low-level function usage in the dataset. For example,

most of the low-level functions are in the constant string LC category, and can hence be replaced using high-level alternatives that directly call the function encoded in the constant string. Similarly, the DS calls can be replaced by the `transfer` function. Together, these two categories constitute 76.9% of the set of all contracts, and 86.5% of the set of unique contracts on Ethereum. If we include the Other category, the percentage of contracts using low-level functions that can be statically replaced increases to 82%.

*Therefore, only* 18.0% *of smart contracts using low-level functions cannot be statically replaced with high-level alternatives (*15.1% IA*s plus* 2.9% *arbitrary* LC*s).* With that said, not all contracts in the remaining 82% are straight-forward to replace, as we describe in the next chapter (Chapter 4).

# Chapter 4

# Methodology

In this Chapter, we present GoHigh, an automated technique to transform the source code of Ethereum smart contracts, and replace low-level functions with their high-level alternatives. GoHigh has three steps. First, it converts the Solidity source codes of smart contract into AST. Then, it searches the AST for 11 patterns of DS and constant string LC, which we extracted from the dataset of contracts containing low-level functions. Finally, GoHigh replaces the matching sub-tree in the AST with high-level alternatives. We start by explaining the challenges in replacing the low-level functions by GoHigh. We then explain each of the above steps, and finally provide an example of how GoHigh works.

## 4.1 Challenge and Contribution

**Challenge:**. Developers tend to use customized checks to prevent the vulnerabilities of low-level functions. For instance, developers may require the return value of an LC to be true, and revert the whole transaction if the call fails. However, these checks may be incomplete or seem right but shadow other vulnerabilities. One general issue is that the patterns do not check the existence of the callee function. If callee functions do not exist, the return value of the LC is always True, which bypasses all the customized checks. Moreover, hard-encoded strings may also be incorrect (e.g., typos, missing parameters, and incorrect encoding).

The main challenge is that one replacement does not work on all patterns of use

of the low-level functions. Figure 4.1 shows an example, where a direct replacement works for the first pattern but does not work for the second pattern. For the first pattern (Fig.4.1a), the replacement strategy is to extract the function name and the parameters and reassemble them into a high-level call. However, if the same replacement strategy is applied to the second pattern (Fig.4.1b), the expression in the if-clause is replaced with a high-level call. Unfortunately, the new if-statement may not pass compilation because the return value of the high-level call may not be a Boolean value (or there may be no return value), and thus cannot serve as the operand of the logical negation operator. Therefore, we need to come up with custom replacement strategies for each pattern.

```
1 address(0x123).call(abi.encodeWithSignature("register(string)", "
    John")); //before
2 address(0x123).register("John"); //after
```

**(a)** A simple pattern and its replacement.

```
1 if(!address(0x123).call(abi.encodeWithSignature("register(string)
    ", "John"))){revert();} //before
2 if(!address(0x123).register("John")){ //fail compilation
3     revert();} //after
```

**(b)** Applying the direct replacement to a complex pattern.

**Figure 4.1:** One replacement does not apply to every pattern.

**Our Contribution:** We distill the patterns of low-level functions manually from our dataset, so that GoHigh can automatically identify the patterns at the AST level. The 11 patterns are combinations of three representations (shown in Table 4.1) and five patterns (shown in Table 4.2). Three of the five patterns exist in all three representations, resulting in nine combinations; two of five patterns exist only in the send representation, resulting in the remaining two combinations.

## 4.2 AST Generation

To extract the patterns, GoHigh first converts the source code into an AST representation using the Solidity compiler, `solc`. An example AST is shown in Fig.4.2. The root node of the tree is a SourceUnit, and its nodes correspond to different syntactic elements in the smart contract.

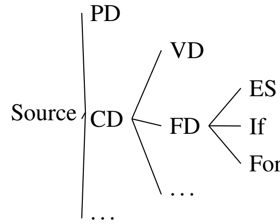


**Figure 4.2:** The structure of an AST of a Solidity smart contract.
∗ PD stands for PragmaDirective, CD stands for ContractDefinition, VD stands for VariableDeclaration, FD stands for FunctionDefinition, and ES stands for ExpressionStatement.

## 4.3 Pattern Matching

**Table 4.1:** The representations of DS and constant string LC.

| Category | Name | Example |
|---|---|---|
| DS | Direct | `addr.send(10ether);` |
| Constant String LC | Direct Encoded | `addr.call(bytes4(bytes256(keccak256("register(string)", "John"))));` |
| Constant String LC | ABI Encoded | `addr.call(abi.encodeWithSignature("register(string)", "John"));` |

GoHigh searches the AST extracted in the previous step for known patterns of low-level functions that we manually extracted from the dataset. To extract the patterns of low-level functions, we first find all possible representations of DS and constant string LC (as mentioned in Section 3.3, we focus on these two categories of low-level functions). We present examples of these representations in Table 4.1. We find that there are two methods to encode constant string LCs in our dataset. The first method, "direct encode", encodes it manually by calculating the keccak256 digest of the function name, and truncating the bytes after the 4 leading bytes. The second method invokes a built-in function, `abi.encodeWithSignature`, and thus we call this method "ABI encode". Solidity also provides other encoding functions, e.g., `abi.encode` and `abi.encodeWithSelector`; however, only `abi.encodeWithSignature` encodes the function signature, while the

others work with a short byte format of function signature (known as "selector" in Solidity). On the other hand, there is only one method for DS, and we call this "Direct".

Second, we iteratively distill the patterns in our dataset of DSs and constant string LCs. For each contract, we write a regular expression to match the low-level functions. The regular expression is as narrow as possible to avoid capturing other patterns. We then remove all contracts that match the regular expression, and repeat this process until there is no contract left. Thus, we formulate regular expressions for the patterns of low-level functions.

Finally, we inspect the above regular expressions to condense them into a smaller set. To do so, we develop ASTs for the patterns distilled in the previous step. We then identify similar AST pairs and attempt to merge them into a new AST. For example, the second example shown in the Require pattern in Table 4.2 adds a string node (message) to the first example's AST. We group these two patterns together due to their similarity. We iteratively repeat the process until we converge. Finally, we obtain the patterns shown in Table 4.2. GoHigh looks for these patterns in the AST of the extracted contract, and performs targeted replacements of the same.

## 4.4 Replacement

Table 4.2 shows the replacement for each pattern identified in the previous section, in terms of the AST transformation. Each pattern has a custom replacement based on its AST. For example, though the if-clause pattern and the if-not pattern both protect the statements located in the if block, they require different replacement patterns as their behaviors differ.

Fig. 4.3a shows an example of if-not pattern, while Fig. 4.3b shows an example of if-clause pattern. Both examples call the `register()` function and roll back if the call fails. In addition, the if-clause pattern performs extra operations (e.g., sending receipt to the caller) in the if statement, which must be preserved after the replacement. Therefore, we keep the if statement in the if-clause pattern, and only replace the expression with a Boolean value True. The call to register is moved to the line before the if statement. However, in the if-not pattern, the operations after

a successful call are located outside of the body of the if clause. Thus, there is no need to preserve any statements after the replacement.

```
1 if(!address(0x123).call(abi.encodeWithSignature("register(string)
    ", "John"))) revert(); //before
2 address(0x123).register("John"); //after
```

**(a)** The If-not pattern.

```
1 if(address(0x123).call(abi.encodeWithSignature("register(string)"
    , "John"))) send_receipt();
2 else revert(); //before
3 address(0x123).register("John");
4 if(True){send_receipt();} else{revert();} //after
```

**(b)** The If-clause pattern.

**Figure 4.3:** Difference between if-not and if-clause pattern.

**Table 4.2:** The patterns of DS and LC for replacement, and the percentages of their occurrence in the base dataset.

| Exists In | Name | %age | Example | AST Representation | Replacement |
|---|---|---|---|---|---|
| Both | Stand Alone | 21.7%∗ | 1 `address(0x123).send(10ether);`<br>2 `address(0x123).call(abi.encodeWithSignature("register(string)", my_name));` | FD — …, LL† | FD — …, HL‡ |
| Both | If Clause | 41.1% | 1 `if(address(0x123).send(10ether)){...}`<br>2 `if(address(0x123).call(abi.encodeWithSignature("register(string)", my_name))){...}` | FD — …, if (LL, block) | FD — …, HL, block |
| DS Only | If Not Clause | 15.4% | 1 `if(!address(0x123).send(10ether)){revert();}`<br>2 `if(!address(0x123).send(10ether)){throw;}` | FD — …, if (cond (NOT, LL), block) | FD — …, HL |
| DS Only | Require | 26.3% | 1 `require(address(0x123).send(10ether));`<br>2 `require(address(0x123).send(10ether), "ERROR_MESSAGE");`<br>3 `assert(address(0x123).send(10ether));` | FD — …, require - LL | FD — …, HL |
| Both | Return | 2.4% | 1 `return address(0x123).send(10ether);`<br>2 `return address(0x123).call(abi.encodeWithSignature("register(string)", my_name));` | FD — …, return - LL | FD — …, HL, return - True |

∗They do not add up to 100% as some contracts have multiple patterns.
†LL stands for low-level function.
‡HL stands for high-level alternative to LL.

To perform the replacement, we perform a level order traversal of the AST, which is a breath-first traversal of the tree. For each expression node, the function iteratively replaces the node or the sub-tree with the high-level expression node until it reaches the leaf node.

## 4.5 Implementation

We implemented GoHigh using a JSONPath-NG package, and SIF [68], a framework of contract instrumentation, to decompile AST form of the contract into its source code form. We have made *GoHigh's source code publicly available*[1].

## 4.6 Example

We provide an example to demonstrate how our replacement works - the source code is shown in Fig. 4.4. The example is extracted from a real-world contract called the Buttonwood Agreement [2], which has more than $1,000$ transaction records in Ethereum. The example contains a contract with one vulnerable function. The function `approveAndCall` can increase a user's allowance and send it a message via `receiveApproval` function, whose address and function name are hard coded into the `_spender` variable (Line 5) and the payload (Line 6). Then, an LC is used in the require statement in Line 6. At the end of the function, the return statement returns True if the LC is correctly invoked. Thus, there is a constant string LC in Line 6 of the contract.

```
1 contract StandardToken is Token {
2     function approveAndCall(address _spender, uint256 _value,
          bytes _extraData) returns (bool success) {
3         allowed[msg.sender][_spender] = _value;
4         Approval(msg.sender, _spender, _value);
5         require(_spender.call(
6             bytes4(bytes32(sha3("receiveApproval(address,uint256,
                  address,bytes)"))), msg.sender, _value, this,
                  _extraData));
7         return true;}}
```

**Figure 4.4:** The example before replacement.

[1] https://github.com/DependableSystemsLab/GoHigh.

[2] 0x2a6a1521a43601c847dd853cbc5f25d5d6505dad

The AST of the example is shown in Fig. 4.5a. The root node is the contract (Source). It has only one child node, `approveAndCall` (FunctionDefinition, FD), because it has only one member function. There are four children of the function declaration: value assignment node (VA), approval event node (AE), require statement node, and return node.

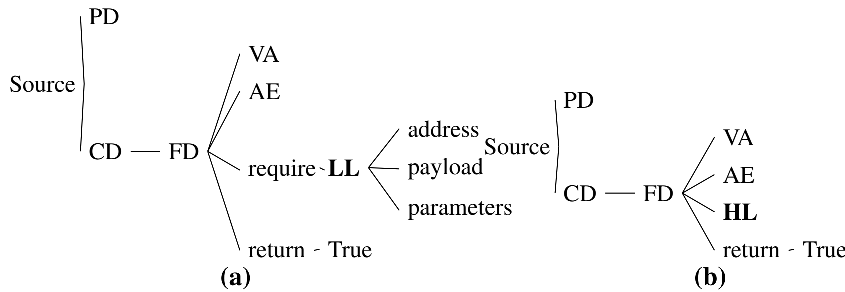


**Figure 4.5:** Example's AST (a) before replacement, and (b) after replacement.
*LL stands for the low-level function and HL stands for the high-level alternative.

After obtaining the AST, we traverse it from the root node, and check if the tree matches the patterns listed in Table 4.2. In the example, the require statement matches the require pattern of constant string LC. Therefore, GoHigh extracts the address (`_spender`), the payload and the parameter list of the low-level function. The extracted components are then reconstructed into a new statement that uses high-level function as per Table 4.2. After replacement, the AST of the example is shown in Fig. 4.5b. Finally, the replaced AST is converted to the source code shown in Figure 4.6. The high-level alternative is in Line 5.

Note that the above replacement leads to a gas cost reduction during contract execution. The original version shown in Fig. 4.4 costs 24,491 gas units, while the replaced version in Fig. 4.6 costs only 24,262 gas units, which is 229 gas units less than that of the original.

## 4.7 Summary

GoHigh provides a pattern-based automated replacement to capture all low-level function representations in different patterns. The AST representation condensed from the regular expressions ensures the accuracy of the capture. Moreover, Go-

```
1 contract StandardToken is Token {
2     function approveAndCall(address _spender, uint256 _value,
        bytes _extraData) returns (bool success) {
3         allowed[msg.sender][_spender] = _value;
4         Approval(msg.sender, _spender, _value);
5         ITokenReceiver(_spender).receiveApproval(msg.sender,
            _value, this, _extraData);
6         return true;}}
7 interface ITokenReceiver{
8     function receiveApproval(address a,uint256 b,address c,bytes
        d);}
```

**Figure 4.6:** The example after replacement.

High performs the replacement in the expression level by leveraging the AST subtree, which guarantees the replacement does not break the syntax correctness of the replaced smart contracts. In Chapter 5, we verify our observations derived in Chapter 3.2 and evaluate the effectiveness of GoHigh on two more recent datasets.

# Chapter 5

# Results

In this chapter, we discuss the experimental setup, followed by the research questions (RQs) we ask. Then, we present the results of the experiments to answer the RQs.

## 5.1 Experimental Setup

To demonstrate the experimental setup, we first present the datasets we used in the experiments, followed by the procedure of processing them. Then, we present the hardware and software we use to collect the datasets and conduct the experiments.

### 5.1.1 Datasets

Because our technique operates at the source code level, we need contracts that are open-source. However, not all smart contract addresses have their source code available in the Ethereum. To obtain the open-source dataset, we followed a two-stage process. First, we obtained a full list of smart contract addresses from a local Ethereum full node with a Ethereum node query package, Ethereum-ETL [1]. Next, we queried the source code from Etherscan [2] using its public API.

We used three non-overlapping datasets using the above method for our experiments, as shown in Table 5.1 [1]. The first is the dataset that we analyzed in Chapter 3.2, which has all the contracts on Etherscan from March 1, 2016 to September 30,

[1] We have made all datasets publicly available in [91].

2019. We refer to this as "base dataset". This set includes $149,363$ open-source smart contracts from a total $17,919,421$ smart contracts deployed in Ethereum in this time period (0.83%).

The second dataset has the contracts created between October 1, 2019, and December 31, 2020. We refer to this dataset as "recent dataset". This dataset consists of $170,304$ contracts. The total number of contracts deployed on Ethereum in this time period is $18,736,340$, in which 0.91% of contracts are captured. *Note that the contracts in the recent dataset were released almost a year after the Solidity guidelines were finalized in November 2018.*

The third dataset includes the contracts created between January 1, 2021, and December 31, 2021. We refer to this dataset as "latest dataset". This dataset consists of $1,831,971$ contracts out of $11,213,714$ smart contracts deployed in Ethereum in this time period (16.33%). *The contracts in the latest dataset were released two years after the Solidity guidelines were finalized.*

In the recent dataset and the latest datasets, we observed that Etherscan introduced JSON format in their source code upload, and so we implement a unified source code parser for both plain text format and JSON format. While only 800 contracts (5%) in the recent dataset are in the JSON format, there are more than 10,000 contracts in the latest dataset in the JSON format. Omitting these contracts will thus affect the comprehensiveness of our study.

From the above numbers, we can observe an increasing trend in both the number of total contracts deployed on Ethereum, and the number of contracts on Etherscan (i.e., Ethereum contracts whose source code is available). The average number of contracts per year deployed from 2016 to 2019 is about 5 million. In 2021, the number of contracts deployed increased to 11 million, which is more than double that of previous years. A similar trend is observed on Etherscan, where the number of contracts more than doubled in this time period. We also observe that the source code availability increased from less than 1% in the base and recent datasets to 16.33% in the latest dataset. This is because the `UpgradeableProxy` contract has been widely used in 2021 to upgrade existing smart contracts in the Ethereum. An example is given in Fig.5.1 - we elaborate on it in the experiment of RQ1.

We also collect the Ethereum transactions on the set of contracts used in all three datasets, from March 1, 2016 to December 31, 2021. We extract the subset

Table 5.1: The datasets used in the experiments.

| Name | Date | Compiler Versions | No. of Contracts | No. of All Contracts |
|---|---|---|---|---|
| Base | Mar 1, 2016 - Sep 30, 2019 | 0.4.0 - 0.5.4 | 149,363 | 17,919,421 |
| Recent | Oct 1,2019 - Dec 31,2020 | 0.4.16-0.7.2 | 170,304 | 18,736,340 |
| Latest | Jan 1,2021 - Dec 31,2021 | 0.3.2-0.8.9 | 1,831,971 | 11,213,714 |

of the transactions whose destination address matches a contract's address in any of our datasets. These transactions are used to determine if the replacement by GoHigh had any unintended changes to the contract's behavior.

### 5.1.2 Hardware and Software

We run all experiments on an Intel(R) Core(TM) i7-7700 @ 3.6GHz processor with 32 GB of RAM running Ubuntu 20.04 LTS. The Solidity compilers we use are downloaded from the official website[2]. We also use a private Ethereum blockchain node for testing. This node uses Geth 1.10.9-stable. The EVM version is Byzantium.

## 5.2 Research Questions (RQs)

We pose six RQs that we attempt to answer with our evaluation:

- **RQ1**: How do the characteristics of the recent datasets in terms of low-level functions compare to the base dataset?
- **RQ2**: What is the coverage of GoHigh in replacing the low-level functions in all datasets?
- **RQ3**: How many compiler warnings are eliminated in the contracts due to GoHigh's replacement?
- **RQ4**: Does the replacement by GoHigh introduce any unintended side-effects in the contracts?
- **RQ5**: What's the change in gas cost of the contracts after replacement with GoHigh?

[2]https://binaries.soliditylang.org/

- **RQ6**: What's the time taken by GoHigh to perform the replacements?

To answer **RQ1**, we repeat the analysis in Section 3.2 on the recent and the latest datasets, and compare the results with those we obtained from the base dataset. As before, we considered both the overall contracts and the unique contracts in both datasets.

To answer **RQ2**, we first run GoHigh on the contracts, and then run a substring matching script (derived from Table 4.1) on the contracts replaced by GoHigh to determine how many low-level functions are still left in the contracts after replacement. We measure the coverage of GoHigh on the datasets (i.e., how many contracts it was able to replace successfully).

To answer **RQ3**, we compile the contracts after they have been replaced with GoHigh in all datasets. We then collect the compiler warnings, and determine if there are differences between these and the warnings collected when compiling the original contracts (we remove the original contracts that do not compile even before replacement from the datasets).

To answer **RQ4**, we compare the external states of the smart contract. We first extract the public variables of each contract (i.e., variables declared with either the `public` modifier and `external` modifier) along with its balance to determine the state of the contract. Then, we deploy both the original and replaced contracts on a private Ethereum blockchain node, after removing the original contracts that fail to deploy in our node. Finally, we replay the transactions in the transaction logs, and compare the external states of the contracts with each other. We say a replacement has "succeeded" if the states match each other after the replay, as this suggests that no unintended side-effect was introduced by GoHigh. We measure the success rate for all datasets.

To answer **RQ5**, we use two methods to estimate the gas used by the contract. The first method is static estimation via the Solidity compiler, and the second is runtime estimation based on gas used in transactions. We use both methods to generate a more comprehensive picture of the influence on gas introduced by GoHigh. The static estimation calculates the sum of gas used for all operations in a given function, while the runtime estimation records the gas used for each transaction that we replay in **RQ4**. We use the average gas used per function to calculate the

change in gas consumption due to the replacement by GoHigh.

Finally, to answer **RQ6**, we measure the minimum, maximum, and average times taken by GoHigh for all datasets.

## 5.3 Results

We organize the results by the RQs.

### 5.3.1 RQ1: Observations from Recent and Latest Datasets

**Table 5.2:** The distribution of low-level function.

| Name | Use Low-level Function | Replaceable Low-level Function |
|---|---|---|
| Base | 13.8% | 82.0% |
| Recent | 37.8% | 77.0% |
| Latest | 40.9% | 96.9% |
| Weighted Average | 40.0% | 95.1% |

The results of RQ1 are summarized in Table 5.2. We report the weighted average distribution of low-level function usage and replaceable low-level function over all datasets by $\bar{X} = \sum_{i \in D} W_i X_i$, where $D$ denotes three datasets, $W_i$ denotes the percentage of contracts of the corresponding dataset, and $X_i$ denotes the percentage in low-level function usage of the corresponding dataset. *Overall, we find that 40% of the contracts use low-level functions, and that 95% of the uses are gratuitous, and hence replaceable.*

We observe that the percentage of contracts that contain low-level functions in the recent dataset is 37.8% (shown in Fig.5.2a), which is 24.0% higher than that the base dataset. The percentage of unique contracts also increases from 17.2% in base dataset to 23.5% in the recent dataset, as shown in Fig.5.2b. In the latest dataset, we observed that the percentage of contracts that contain low-level functions is 40.9% (shown in Fig.5.4a), which is 3.1% higher than that the recent dataset. Yet, the percentage of unique contracts decreases from 23.5% in base dataset to 11.9% in the recent dataset, as shown in Fig.5.4b. *Thus, low-level functions are actually increasing in number in both the recent and latest datasets despite the publication of the guidelines.*

Fig.5.3 shows the distribution of the low-level functions in the recent dataset. The most significant change is that the major category changes from constant string LC to DS in the recent dataset (at 71.7%). An in-depth analysis of the contracts in the recent dataset reveals that one-fourth of the contracts are `Forwarder` contracts and `Proxy` contracts. Figure 5.1 shows examples of a `Proxy` contract (Fig. 5.1(a)) and a `Forwarder` contract (Fig. 5.1(b)). The `Forwarder` contract serves as a payment forwarder to a user's wallet (Line 3), and contain DSs. Further, many of the contracts are `Proxy` contracts. The `Proxy` contract uses arbitrary LCs (Line 3) to dynamically invoke functions defined in the `implementation` [62]. It is used to redirect function calls as it is not possible to upgrade a smart contract once it is deployed on the blockchain [40]. This is why DS and arbitrary LC are more prevalent than constant string LC in the recent dataset. *In total, GoHigh can still replace more than 75% of the low-level functions in the recent dataset (*71.7%DS + 4.9%*CS*LC + 0.4%*Other* = 77.0%*).*

Fig.5.5 shows the distribution of the low-level functions in the latest dataset. Compared with the recent dataset, the major category changes from DS to constant string LC (at 90.4%)., which is aligned with the observation from the base dataset. However, it does not mean that the use of constant string LC in the latest dataset is the same as that in the base dataset. A deep dive into the latest dataset reveals that the majority of the constant string LC contracts are `Forwarder` contracts. We term these `ForwarderERC20` contracts. Fig. 5.1(c) shows an example of a typical `ForwarderERC20` contract. Different from `Forwarder` in Fig. 5.1(a), `ForwarderERC20` contract forward the ERC-20 token [85] transfer instead of ETH transfer. It hard-codes the signatures of ERC-20 interfaces (e.g., `transfer()`) (Line 5), and thus is classified into our constant string LC category. This also explains why constant string LC is the most prevalent in the latest dataset. *We can draw a similar conclusion that GoHigh can still replace more than 96% of the low-level functions in the latest dataset (*6.5%DS + 90.4%*CS*LC = 96.9%*).*

### 5.3.2 RQ2: Coverage of GoHigh's replacement

All the DS and constant string LC contracts in all three datasets are correctly identified by GoHigh. After replacement by GoHigh, none of the target contracts con-

```
contract UpgradeableProxy {
    function _delegate(address implementation, bytes data)
        internal {
        (bool res, bytes returnData) = implementation.
            delegatecall(data);
        require(res); return returnData;}}
```

**(a)** `Proxy` contract

```
contract Forwarder {
    address public destinationAddress;
    receive() public payable {destinationAddress.send(msg.value)
        ;}}
```

**(b)** `Forwarder` contract

```
contract ForwarderERC20 {
    address public destinationAddress;
    function flushTokens(address ERC20TokenAddress, uint amount)
        public {
        ERC20TokenAddress.call(
            abi.encodeWithSignature("transfer(address,uint256)",
                destinationAddress, amount));}}
```

**(c)** `ForwardERC20` contract

**Figure 5.1:** Examples of (a) `Proxy` contract, (b) `Forwarder` contract and (c) `ForwardERC20` contract.

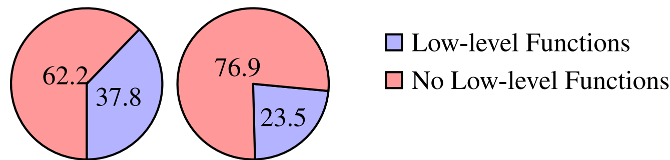


**Figure 5.2:** Low-level functions usage in the **recent** dataset for (a) all contracts and (b) unique contracts.

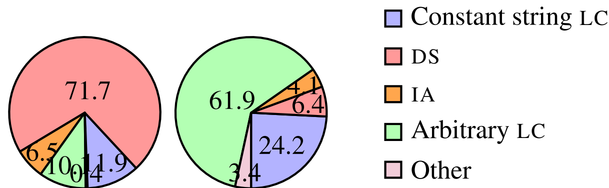


**Figure 5.3:** The distribution of low-level functions in the **recent** dataset for (a) all contracts and (b) unique contracts.

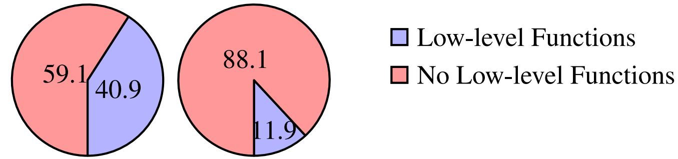


**Figure 5.4:** Low-level functions usage in the **latest** dataset for (a) all contracts and (b) unique contracts.

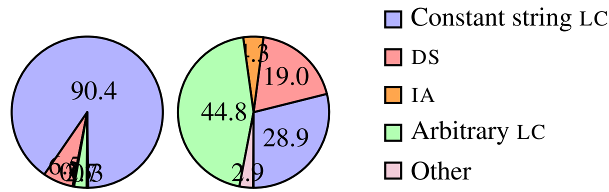


**Figure 5.5:** The distribution of low-level functions in the **latest** dataset for (a) all contracts and (b) unique contracts.

tain either DS or constant string LC low-level functions. While this result is not surprising for the base dataset (as we used it to identify the replacement patters in Table 4.2), the recent dataset and the latest dataset were also completely covered by the patterns identified from the base dataset despite neither of them being used for pattern extraction. *Therefore, GoHigh has an overall coverage of* 100% *in identifying the patterns of low-level functions in both the recent and latest datasets*. Recall from RQ1 that this constitutes 82% of the contracts using low-level functions in the base dataset (Section 3.2), 77% in the recent dataset, and 96.9% in the latest dataset(RQ1).

Note that we observed that six contracts in the recent dataset used “send” even after replacement. However, these are not missed cases because the “send” function is overridden in their code, and thus they do not belong to the DS category. GoHigh correctly identified these cases and did not perform the replacement.

### 5.3.3 RQ3: Compiler Warnings before and after Replacement

Table 5.3 shows the results of RQ3. We report the weighted average distribution of compiler warnings changes over all datasets in the last column using the same equation in RQ1. *Overall, we find that GoHigh removes warnings in 4.9% of the*

**Table 5.3:** The distribution of complier warnings change before and after replacement.

| Name | Unchanged | Remove | Add |
|---|---|---|---|
| Base | 82.1% | 16.4% | 1.5% |
| Recent | 80.5% | 17.6% | 1.9% |
| Latest | 96.2% | 3.7% | 0.1% |
| Weighted Average | 94.8% | 4.9% | 0.3% |

*smart contracts.*

In the base dataset, all contracts passed compilation after replacement by GoHigh. Figure 5.6 shows the changes in warnings for contracts using DS and constant string LC before and after replacement in the base dataset. As can be seen, the majority of compiler warnings are unchanged as they do not pertain to low-level functions. Further, there is a 16% reduction in warnings due to the replacement of low-level functions (shown as "Remove" in the figure).

For example, the compiler expects developers to check the return value of LCs, and will output a warning if the check is missing. Because GoHigh replaces LCs, the compiler will not emit this warning. Similarly, using DS anywhere in the contract results in a warning. GoHigh removes all DSs in contracts, and so the warnings due to the use of DSs are also removed.

However, a few new warnings are added due to the replacement of low-level functions in some cases (shown as "Add" in the figure). Most of these are due to unused variables in the contract after the replacement. For example, in Fig. 5.7, replacing the LC does not remove the declaration of the Boolean variable `result` in Line 5. However, these are benign warnings as they affect neither the contract's correctness nor its security.

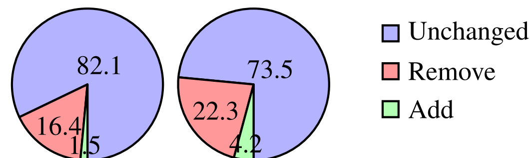


**Figure 5.6:** Change in compiler warnings for the base dataset for (a) DS, and (b) Constant String LC.

```
bool result;
result = address(0x123).call(
    abi.encodeWithSignature(
    "register(string)", "John")); //before
bool result; //warning: Variable declared but unused
address(0x123).register("John"); //after
```

**Figure 5.7:** Warning Added after replacement by GoHigh.

For the recent dataset, 99.9% of the DS contracts and 95.4% of constant string LCs successfully passed compilation after replacement by GoHigh. It is not 100% due to a recently introduced modifier in Solidity regarding the storage location of dynamic-sized arrays (e.g., string, byte array). Our analysis does not currently handle this modifier, and hence the replacement by GoHigh results in compilation errors.

The results of the changes in compilation warnings for the recent dataset are shown in Fig.5.9. As can be seen, the reductions in warnings for contracts using DS is consistent with that of the base dataset, i.e., 17% warning decrease. However, there is a 38.2% decrease in the compiler warnings in the constant string LC contracts, which is 16% higher than the decrease in warnings in the base dataset. This is because Solidity version 0.5.0 disallows the use of `sha3` function (a hash function) in favour of `keccak256` function [14], and using `sha3` function results in compiler warnings. The `sha3` function and `keccak256` function are used to manually encode the function signatures in constant string LCs (see "Direct Encoded" in Table 4.1). The removal of constant string LCs by GoHigh removes warnings on using the `sha3` function.

```
// solhint-disable-next-line avoid-low-level-calls
(bool result,) = to.call(abi.encodeWithSignature("register(string
    )", name));
```

**Figure 5.8:** An example on special comment that disables low-level calls.

For the latest dataset, 97.1% of the DS contracts and 96.2% of constant string LCs successfully passed compilation after replacement by GoHigh. It is not 100% due to the same issue as that with the recent dataset. Figure 5.10 shows the results of the changes in compilation warnings for the latest dataset. The reduction in warnings for contracts decreases in both DS and constant string LC contracts,

which are 3.7% and 6.7%, respectively. Compared with the reduction in warnings in the base and recent datasets, the reduction in warnings in the latest dataset decreases by about 10%. This is because developers intentionally disable warning output via special comments in the source code. Figure 5.8 shows an example of a special comment that disables compiler warnings on using low-level calls. With this special comment, even though GoHigh replaces DS and LC, the warnings are suppressed, and so this case is included in the "Unchanged" category (blue part).

*Overall, GoHigh significantly decreases the number of compilation warnings in the smart contracts by 4.9% after replacement.*

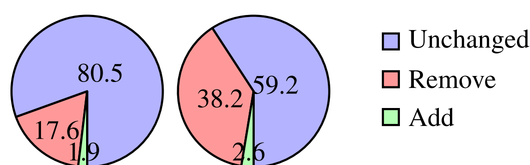


**Figure 5.9:** Change in compiler warnings for the recent dataset for (a) DS, (b) Constant String LC.

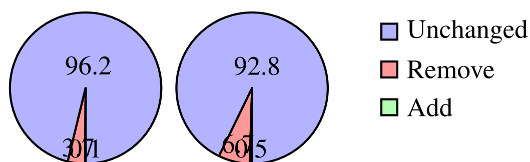


**Figure 5.10:** Change in compiler warnings for the latest dataset for (a) DS, (b) Constant String LC.

### 5.3.4 RQ4: Unintended Side-Effects After Replacement

Table 5.4 shows the results of transaction replay for the three datasets. As before, we report the weighted average success rate using the equation in RQ1. The weighted average success rates of the base dataset, the recent dataset and the latest dataset are 95.1%, 92.4%, and 79.8%, respectively. The decrease in the success rate across three datasets is mainly due to the increase of external contract dependency in Solidity smart contracts, which we explain in the discussion to RQ4.

These were all due to cases where we could not verify if the replacement succeeded, and hence could not label it as a success. However, we did not observe any case for which there was a state mismatch after the replacement.

**Table 5.4:** The success rate in replaying transactions to replaced contracts.

| Name<br>Category | Base Dataset<br>DS CSLC | Recent Dataset<br>DS CSLC | Latest Dataset<br>DS CSLC | Weighted Average<br>DS CSLC |
|---|---|---|---|---|
| Success Rate (%) | 98.7% 94.1% | 68.6% 96.5% | 94.6% 70.5% | 92.8% 72.4% |
| Weighted Average Success Rate (%) | 95.1% | 92.4% | 79.8% | 80.6% |

The reason why some contracts do not succeed is that their transactions use arbitrary calls to functions defined in external contracts. We cannot confirm that the state matches for these contracts if the external contracts modify the state of the tested contracts in arbitrary calls, as we cannot deploy the external contracts on our private blockchain (because it is non-trivial to decode their address based on the call and to replicate them). We manually investigated all the contracts in which GoHigh did not succeed, and found that all of them fall into this category. Thus, there was no contract for which there was a state mismatch.

Overall, none of the contracts in which GoHigh performed replacements failed to match the states of the original contracts, for the replayed transactions. However, some of them could not be verified due to external dependencies.

*The success rate of GoHigh is 80.6% across the three datasets.*

```
1 contract Avatar {
2     function externalTokenTransfer(IERC20 _externalToken, address
          _to, uint256 _value)
3     public onlyOwner returns(bool){
4         address(_externalToken).safeTransfer(_to, _value);
5         emit ExternalTokenTransfer(address(_externalToken), _to,
              _value);
6         return true; }}
```

**Figure 5.11:** An example of a Solidity smart contract that contains external dependency.

**Why replaying is difficult for contracts with external dependencies?** Replaying an existing transaction on the Ethereum mainnet (the primary public Ethereum

production blockchain) is difficult for contracts with external dependencies. To elaborate further on the difficulty, an example is given in Fig. 5.11. The example is a real-world contract[3] in our dataset, named `Avatar`, which is a reputation management system. The goal of the function `externalTokenTransfer()` is to transfer ERC-20 token (defined in `externalToken`) to a given address (defined in `to`).

First, we try to directly replay the original transaction without other settings, i.e., pass the same ERC-20 token address (`externalToken`) and receiver address (`to`). The transaction reverts because of the `onlyOwner` modifier. It requires that the caller of the function is the same account as its owner. Then, we define another owner address in our local Ethereum network and deploy `Avatar` again, and we replay the transaction using the new owner's address. The transaction passes the `onlyOwner` check, but it reverts again because of the missing `externalToken` implementation in the given address. Recall that we are running the experiments on our local network which has no `externalToken` implementation. When the EVM tries to look for the `safeTransfer()` method defined in `externalToken`, the result is empty and thus EVM revert the execution.

Next, we deploy the missing `externalToken` implementation to the local network. The implementation can be retrieved from the contract creation transaction, i.e., the first transaction of `externalToken` contract. We replay the original transaction again. With the implementation of `externalToken`, EVM successfully finds the `safeTransfer()` method. However, calling the method reverts the execution because the token sender, i.e., contract `Avatar`, does not have sufficient amount of token to transfer. The token amount of all token holders are kept in `externalToken` contract's state. In the Ethereum mainnet, the token may be sent from another EOA or contract in a separate transaction. However, in the local net, the token is zero because we have not replayed the initial token transfer.

Finally, if we can find the initial token transfer transaction, we can process the original transaction. However, the token transfer may be from any source. For

[3] 0xf8aeeac857ccb59020acc821aa27cad92765addd

example, it can be transferred from another contract or be minted from the user. Unfortunately, we have no indication of where to find the initial token transfer transaction.

The above example shows that it is difficult to resolve the reverts in replaying transactions that have external dependencies. What is more, there are no existing tools that provide seamless transaction replay functionality for contracts that have external dependencies. *Therefore, we skip the validation of these 19.4% of contracts because of their dependencies on the external contracts and the underlying transactions, and hence the overall success rate is 80.6%.*

### 5.3.5 RQ5: Gas Cost Overhead of GoHigh

**Table 5.5:** The gas used overhead introduced by GoHigh.

| Name | Static Estimation Increase(%) | Avg. Change | Runtime Estimation Increase(%) | Avg. Change |
|---|---|---|---|---|
| Base | -1.81* | -176 | -3.42 | -29,823 |
| Recent | +0.24 | +19 | -0.00 | -1 |
| Latest | -5.10 | -431 | -5.74 | -17,869 |
| Weighted Average | -4.69 | -396.96 | -5.32 | -16,961.23 |

*Negative value in overhead means decrease in gas used, which means the GoHigh saves gas.

Table 5.5 shows the average gas changes (both as percentages and absolute values) using the two gas estimation settings for each of the datasets. A positive value indicates that GoHigh increases the gas consumption of the contract, while a negative value indicates that GoHigh decreases the gas consumption. We report the weighted average gas saving both in percentage and in gas units using the equation in RQ1.

*We find that GoHigh is able to reduce the gas consumption by 4.69% based on static estimates, and by 5.32% based on runtime estimates across the three datasets.* As can be seen, in the majority of cases, the gas used decreases after replacement by GoHigh. The highest decrease is recorded in the latest dataset, which has more than 5% gas savings in both static and runtime estimation.

The gas reduction achieved by GoHigh is due to pruning unnecessary `require` clauses and `if` clauses in the function body, which then removes the `JUMP`, `EQ` op-

eration in the EVM opcodes that consume gas. The only increase in gas consumption is in the static estimation of the recent dataset. However, the run time estimate on the same dataset shows that the gas consumed decreases. The contradiction may result from the redundant returns introduced by GoHigh. Recall that the replacement of Return pattern (in Chapter 4), the replacement may add `return true;` to match the interface after replacement. As a result, it introduces more operations to the contract, and leads to a small gas overhead in statics estimation. However, the Return pattern function is not executed frequently (at 2.4%), and hence there is a net decrease in the runtime estimation.

We find the overall gas used decreases after GoHigh's replacement, regardless of whether we use static or runtime estimation. *The average gas used reduces over 5% ($-5.01\% = (-4.69\%\ static + -5.32\%\ runtime)/2$ estimations) after replacement.* This shows that replacement of low-level functions is actually beneficial for gas consumption (in most cases).

**Why use two methods for gas estimation?** We use two methods for gas estimation as there are different advantages and drawbacks of each technique. The static estimation provides an accurate theoretical lower bound of the gas used in a function. The rationale is that static estimation calculates the gas used by examining the compiled opcodes of the smart contract, which guarantees that the estimation does not miss any operations in the contract. However, a static estimation is often inaccurate as it fails to handle dynamic gas usage, i.e., loop over a dynamic array.

Figure 5.12 shows a smart contract, `Swapper`[4] that has a function with loop. The for-clause iterates over a dynamic array `commission`. Given that the length of `commission` is determined on the fly, the static estimation cannot give an accurate estimate of the gas used. If the user passes a long commission array into this function, it will lead to significantly higher gas consumption than what was statically estimated.

On the other hand, the runtime estimation is based on the real-world transactions that are executed on the Ethereum mainnet, and thus reflects a more comprehensive view of the gas used in real user behaviors. However, the shortcoming of

[4] 0xfb774ac09c2cb7a6f47b4d8367293820e4b4c660

```
function takeCommission(Swapper swapper, address token,
    Commission[] memory commission
    ) internal onlyAdmin() {
        for (uint i = 0; i < commission.length; i++) {
                require(swapper.getBalance(token) > commission[i
                    ].amount, 'Swapper balance not enough for
                    commission');
                swapper.claim(commission[i].destination, token,
                    commission[i].amount);}}
```

**Figure 5.12:** An example when Solidity compiler cannot estimate the actual gas used.

runtime estimation is the bias against infrequently-used functions. For example, contract executions to privileged functions are less frequent in transactions of a contract. Therefore, the gas overhead of privileged functions introduced by GoHigh is less significant to the commonly-used functions. This is reflected in the lower gas costs of GoHigh's replacement due to runtime estimation.

Therefore, we use both static and runtime estimation for understanding the gas cost of GoHigh's replacement.

### 5.3.6 RQ6: Performance of GoHigh

The average time taken by GoHigh to perform replacement in a smart contract is 6.59 seconds across all datasets. The minimum time is 0.11 seconds, and the maximum is 60 seconds. Overall GoHigh had a replacement time of under 10 seconds for more than 80% of the contracts across all the datasets. Figure 5.13 shows the histogram of the time taken by GoHigh across smart contracts.

## 5.4 Summary

In this chapter, we describe how low-level functions are used in two more recent datasets, i.e., the recent dataset, and the latest dataset, followed by the evaluation of the effectiveness, gas efficiency, and performance of GoHigh's replacement. We employ the compiler outputs of both original smart contracts and their replaced ones to evaluate GoHigh's effectiveness in eliminating compiler warnings. Furthermore, we replay the historical transactions on Ethereum main net to verify whether

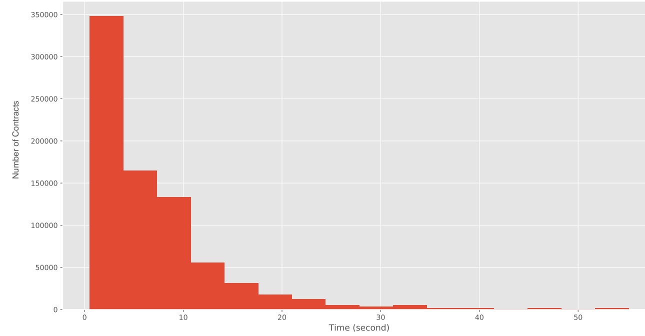


**Figure 5.13:** The histogram of GoHigh's performance across smart contracts.

GoHigh's replacement leads to unintended changes. Finally, we estimate the gas cost of original smart contracts and their replaced ones and calculate GoHigh's runtime performance.

We demonstrate the effectiveness, gas efficiency, and performance of GoHigh, and the main results are as follows: (a) 40% of the smart contracts still use low-level functions, and 95% of the uses are gratuitous and replaceable; (b) GoHigh achieves 100% coverage in identifying the patterns of low-level functions; (c) GoHigh reduces 4.9% of the compiler warning after replacement; (d) we verify 80.6% of the replaced smart contracts and confirm that no unintended side-effects are introduced by GoHigh- the remaining 19.4% are not verifiable due to their external dependency; (e) GoHigh reduces the gas cost by 5% on average after replacement; and (f) GoHigh takes an average of 6.59 seconds to perform replacement of a contract. We further elaborate the limitations on our evaluations in Chapter 6.2 and discuss potential reasons why developers violate the guidelines in Chapter 6.

# Chapter 6

# Discussion

In this chapter, we start by speculating possible reasons why developers intentionally violate the guideline and use low-level functions. We then delve into the threat to validity of this thesis.

## 6.1 Possible Reasons Why Developers Violate the Guideline

Our results show that there is widespread use of low-level functions in real-world smart contracts, despite the publication of the Solidity guidelines discouraging their use. Furthermore, we found that many of these uses are gratuitous for the smart contract's functionality, and can be replaced with high-level alternatives, without any change in the contract's functionality or increasing its gas consumption. However, the question remains as to why developers did not follow the guidelines in the first place. In this chapter, we speculate on some of the reasons why developers do not follow the guidelines, based on our results.

### 6.1.1 Gas Consumption Misunderstanding

There is a misunderstanding among programmers that a Low-level Call (LC) and an Inline Assembly call (IA) are more gas-efficient than a High-Level one (HL). However, it is not always the case in Solidity smart contracts. We compare the gas consumption of calling a frequently-used ERC-20 `transfer()` function using

```
1 function highLevelCallTransfer(uint amount) public {
2     token.transfer(dest, amount); }
```

**(a)** High-level Call (HL)

```
1 function lowevelCallTransfer(uint amount) public {
2     (bool res, ) = address(token).call(
3         abi.encodeWithSignature("transfer(address,uint256)", dest
            , amount));
4     require(res, "Transfer failed"); }
```

**(b)** Low-level Call (LC)

```
1 function assemblyCallTransfer(address token_, address dest_, uint
     amount) public {
2     bytes4 sig = bytes4(keccak256("transfer(address,uint256)"));
3     bool res;
4     assembly{
5         let x := mload(0x40)
6         mstore(x, sig)
7         mstore(add(x,0x04), dest_)
8         mstore(add(x,0x24), amount)
9         res := call(gas(), token_, 0, x, 0x44, x, 0x0)}
10    require(res, "Transfer failed"); }
```

**(c)** Inline Assembly call (IA)

**Figure 6.1:** An example of calling ERC-20 `transfer()` in three different methods.

**Table 6.1:** The gas consumption of calling ERC-20 `transfer()` in three different methods.

| Method | IA | LC | High-level Call |
|---|---|---|---|
| Gas Consumption | 56,349 | 43,409 | 43,027 |

three different methods, i.e., LC, IA and HL. Table 6.1 shows the gas used by the methods. The demo code we used is shown in Fig. 6.1. *The result shows that using HL is the most gas-efficient method* in transferring an ERC-20 token, costing only 43,027 units of gas. However, LC costs 43,409 units, which is 0.9% higher than HL. IA is the most expensive method among the three calls, representing 56,349 units of gas, which is 31.0% higher than HL. The experiment is conducted on Remix with a JavaScript EVM emulator with a gas profiler plugin.

The results reveal developers' misunderstanding of the gas consumption of different calls: in fact, LC is not as gas-efficient as HL. Also, the gas efficiency of IA depends heavily on the developer's control over storage. For example, the IA allows

developers to pack input parameters according to the actual width of the variables (e.g., 20 bytes for an address), instead of wasting gas in zero paddings (e.g., there are 12 bytes zero paddings in a 32-byte-width address). Also, the return data of IA can reuse the storage of input data, which avoids wasting gas in extra storage. However, the frequently-used IA pattern (shown in the figure) fails to handle it and thus leads to extra gas consumption. *Only with fine-grain control to storage management can* IA *save gas for its users, otherwise it leads to extra gas consumption. Thus, HL is the most gas efficient solution in practice.*

### 6.1.2 Overly Restrictive Guidelines

The official guidelines are too restrictive, and thus developers have to violate the guidelines to accomplish their desired use cases. One famous example is ETH transfer, and there are three possible ways of transferring Ether, shown in Fig. 6.2, i.e., Deprecated `send` (DS), guideline-recommended `transfer` and community-recommended `call{value: v}("")` (LC).

The guideline-recommended `transfer` is the most secure method to transfer ETH to an EOA because it prevents all kinds of reentrancy attacks by specifying the gas limit to its receiver, which is 2300 units. However, the limit is so low that it cannot support any interaction defined in smart contacts, and thus the receiver can only be an EOA. HTherefore, if some ETH is transferring to a smart contract that has complex calculations, using `transfer` leads to failure and the ETH is not sent. This, the common method for sending ETH is to use LC[1], instead of the documentation-recommended `transfer`. LC forwards all available gas to its destination, and thus supports complex interactions between contracts. Thus, developers intentionally violate the official guidelines to extend the interoperability of their smart contracts.

```
1 to.send(msg.value); //DS
2 to.transfer(msg.value);  //transfer
3 to.call{value: msg.value}(""); //LC
```

**Figure 6.2:** The different methods in transferring Ether.

[1] https://solidity-by-example.org/sending-ether/

### 6.1.3 Vulnerable Access Control Library

```
1 contract WalletLibrary {
2     modifier only_uninitialized() { /* check if uninitialized */
          }
3     modifier onlymanyowners(bytes32) { /* check if signed by
          owners */ }
4     function initWallet(address[] _owners, uint _required, uint
          _daylimit) only_uninitialized {
5         // initialization logics}
6     function kill(address _to) onlymanyowners(sha3(msg.data))
          external {
7         suicide(_to);}}
```

**(a)** A library contract that provides initialization logics with access control.

```
1 contract Wallet {
2     function Wallet(address[] _owners, uint _required, uint
          _daylimit) {
3     bytes4 sig = bytes4(sha3("initWallet(address[],uint256,
          uint256)"));
4     address target = _walletLibrary;
5     uint argarraysize = (2 + _owners.length);
6     uint argsize = (2 + argarraysize) * 32;
7     assembly {
8         mstore(0x0, sig)
9         codecopy(0x4,  sub(codesize, argsize), argsize)
10        delegatecall(sub(gas, 10000), target, 0x0, add(argsize, 0
              x4), 0x0, 0x0)}}}
```

**(b)** A contract that use library function via Inline Assembly call (IA).

**Figure 6.3:** An example of a contract reusing logics in a library contract.

Developers trust the reliability of well-known libraries, and thus delegate calls (using low-level functions) to the libraries and reuse the logic defined in the libraries. Reusing logic in existing libraries saves the gas fee in deployment, so it is becoming a common practice in developing smart contracts. Figure 6.3 shows an example of a smart contract that reuses initialization logic defined in the library. The constructor function of `Wallet` (Fig. 6.3(b) Line 2) use IA to call the `initWallet()` function defined in `WalletLibrary` (Fig. 6.3(a) Line 4).

Some functions in a smart contract are designed to be dangerous, e.g., the suicide function that will empty all the data in the contract address (Fig. 6.3(a) Line 6), the rescue function that will send all the ETH to a given EOA, and the pause function that stops all other functions. These functions can be abused by malicious

users [96], and hence are discouraged by the Solidity guidelines. In practice, however, smart contracts often use access control mechanisms, so that only privileged users have the capability to execute protected functions (e.g., only owners can call `kill()` function in Fig. 6.3(a) Line 6). It means that even though some functions violate guidelines, they are accessible to privileged users only, and thus the smart contracts are secure from attacks.

Unfortunately, the access controls implemented in libraries are not always correct. The example shown in Figure 6.3 is from a real-attack to Parity Multi-Sig library[2]. Although the library protects the `kill()` function with access control, the `owners` field is not protected and thus the attackers can exploit it to kill the library contract. As a result, all the underlying `Wallet` contracts are locked in perpetuity, and can never be accessed.

Thus, developers are able to handle dangerous functions, which are discouraged by the guidelines using access control, and hence they intentionally violate the guidelines to leverage the smart contract's features (suicide, rescue, and pause). Unfortunately, the access controls are not 100% reliable due to subtle bugs in their implementations.

### 6.1.4 Upgradeability and Proxy

Because smart contracts are difficult to upgrade once deployed, developers use the concept of "proxy" to upgrade existing smart contracts. With this mechanism, it is possible for users to specify "the implementation" smart contract address to dynamically invoke different versions of smart contracts. An upgradeable smart contract consists of two parts. The first part is a `Proxy` contract (Fig. 5.1). The `Proxy` contract is the gateway to all functions to the implementation contract, which is the second part of the upgradeable smart contract. Users first pass the address of the implementation contract and the payload to the `Proxy` contract (Line 2). The `Proxy` contract then forwards the payload to the specific implementation (Line 3). The forwarding is made possible by the LCs. Specially, the `delegatecall` LC preserves the original `msg.sender` information in the call stack, and thus it is widely used in upgradeable smart contracts. EVMPatch [74]

[2] 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

also adopts the upgradeable proxy design to migrate the original contract to its patched version.

In addition to violating the guidelines of not using low-level functions, employing an upgradeable proxy design also violates the recommended practice of using the constructor function. A constructor function is a built-in feature of Solidity smart contracts, which allows users to initialize the state variables of a smart contract. However, an upgradeable contract violates the above guideline, since the constructor is bypassed in an implementation contract.

## 6.2 Threats to Validity

In this section, we consider external threats and internal threats to the validity of the experiments.

### 6.2.1 External Threat

**Sampling Bias in Dataset**. An *External* threat to validity is dataset sampling bias as we used only the open-source contracts to perform the evaluation experiments. The total number of contracts submitted to Ethereum is 17 million, but our base dataset has just under $150,000$ contracts, which is less than $1\%$ of all the contracts in Ethereum. However, we do not consider the other contracts as it is non-trivial to determine whether a contract uses a low-level function without having access to its source code. Meanwhile, GoHigh is designed to be a tool for developers, who have the access to the contract's source code. Further, this problem is not unique to GoHigh; for example, prior work SMARTSHIELD [98] has the same issue.

**Verification without External Dependency**. Another *External* threat to validity are external call transactions, which make it infeasible for our verification process to check whether the state changes are the same for the replaced contract as the original one. This is because we run the transactions on a private blockchain, as we do not want to pay gas for running them on the public blockchain. This can be alleviated by extracting these contracts and running them on the private blockchain, but requires manual effort. We manually verify a contract by migrating all external dependency in Chapter 5, however, the migration of a single contract's external dependency took about two days, and about 900 Gigabytes of storage space to

store the Ethereum snapshot, as we demonstrate in Section 5. Expanding the above process to all the unverifiable contracts (at 163,400 contracts) is not practical.

### 6.2.2 Internal Threat

**Differences between EVM Versions**. The first *Internal* threat to validity is the difference in the versions of EVM used for running the experiments. We run all the experiments on the default EVM version of `solc` compiler, Byzantium. However, some errors in deployment resulted from the mismatch of EVM version, as the EVM versions of the contract vary. This issue can be alleviated by using different versions of the EVM, but is logistically more challenging to deploy.

**Inline Assembly Patterns**. The second *Internal* threat to validity is that we do not handle all IA categories. Although we included IA in Chapter 3, we currently do not have a systematic understanding of developers' use of this category. This is because there is wide variation among the different uses of this category, and we hence cannot distill patterns from the datasets to capture IAs.

**Subjective Selection on Warnings from the Solidity Documentation**. The third *Internal* threat to validity is that we filter Solidity documentation warnings subjectively. The warnings' compliance with the criteria for inclusion is based on our best knowledge. For the first and the second criteria, i.e., no mature solutions, and real-world exploits, we examine the warnings based on our best knowledge. Moreover, new solutions and new exploits may emerge in the future, and thus these criteria may result in different warnings being chosen.

For the third criterion, i.e., no semantic knowledge, it is difficult to come up with a clear line between "require semantic knowledge" and "do not require semantic knowledge", because all the expressions in a smart contract serve as a part of the semantic knowledge of the contract. Hence, we cannot formally and quantitatively distinguish the degree of "the requirement of semantic knowledge", and thus do it based on our experience. For example, in one warning on BT, the warning says "Do not rely on `block.timestamp` or `blockhash` as a source of randomness, unless you know what you are doing." Whether or not using BT as a source of randomness requires an understanding of how to use BT in the context of the contract's functionality, so we classify this warning as a "require semantic

knowledge" one. For instance, in an auction contract where its buyers can only place a bid within a given time interval, using a BT is safe since it is not used for generating random numbers. However, classifying whether a given contract is an auction contract requires semantic knowledge, and hence we do not consider BT-related warnings. For other warnings, e.g., LC, DS, and IA, they do not place explicit restrictions on their usage, so we classify them as "do not require semantic knowledge" warnings.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

In this thesis, we defined low-level functions that lead to security vulnerabilities in Ethereum smart contracts written in the Solidity language. We carried out a large-scale empirical study of the use of low-level functions in Ethereum smart contracts. We found that low-level functions are widely used in real-world smart contracts ( 40.0%). Further, we find that more than 95% of the low-level functions are gratuitous for the contract's functionality.

We proposed GoHigh, an automated source-to-source transformation tool for replacing low-level functions in smart contracts with their high-level alternatives. We evaluated GoHigh on three datasets consisting of over $2,100,000$ real-world contracts deployed on Ethereum. The results show that GoHigh is able to *replace all the contracts that contain low-level functions* that are amenable to be replaced, and takes an average time of seven seconds per contract. Further, GoHigh reduces 4.9% of compiler warnings after replacement, and for all contracts that can be verified, the changes introduced by GoHigh do not modify their external behavior. Finally, GoHigh reduces 5% of the gas consumption of the contract after the replacement, and takes 7 seconds on average per contract for the replacement.

## 7.2 Future Work

In this thesis, we find that while most uses of low-level functions are gratuitous, they are important for some specific use cases (e.g., the upgradeable proxy we discussed in Chapter 6). However, the misuse of low-level functions leads to potential vulnerabilities, which can be exploited by adversaries. There are three directions in which this thesis can be extended.

### 7.2.1 Inline Assembly (IA)

We highlight the wide use of vulnerable IA in Solidity smart contract in Chapter 3.2 and Chapter 5. Although IA is one of the low-level functions, we do not extensively study the behavior of IA in this thesis. However, IA plays an important role in Solidity smart contracts, especially in frequently used library contracts. Meanwhile, there has been limited work [48] on IA and its vulnerabilities. One future work direction is to involve interpreting and optimizing the dependability and gas efficiency of IA by translating the IA into its high-level alternatives. This will significantly limit the attackers' capabilities in exploiting the vulnerabilities in IA, as the high-level alternatives benefit from compiler checks.

NLP models can be a good candidate for translating the IA to a more secure form or its high-level alternative. Machine learning techniques have been shown to be effective in translating and generating code [78, 95]. Furthermore, by leveraging the existing dataset, NLP models can be designed to achieve an end-to-end translation, without manually rewriting the source code.

### 7.2.2 Upgradeable Proxy

We show that the upgradeable proxy pattern is widely uses for Solidity smart contract updates, which may introduce vulnerabilities in the migration (e.g., the Parity Wallet library suicide attack in Section 6). One way to mitigate upgradeable proxy vulnerabilities is to automatically migrate existing smart contracts to upgradeable contracts.

Similar to replacing low-level functions, migrating an existing smart contract to an upgradeable contract without breaking its functionality requires non-trivial effort. First, the migration affects all the constructor functions in smart contracts,

which requires developers to rewrite them into `initialize` functions. Second, the implementation contracts must preserve their state variables layout during the migration, otherwise, the upgraded implementation will not work properly.

Hence, one direction of future work is to focus on developing an automated migration technique to upgradeable smart contracts, which preserve the functionality of the constructor function and the state variables layout.

### 7.2.3 Access Control

All the prior work in this domain has been confined to vulnerability detection and testing. Once the vulnerability is detected, the tools throw a warning to the developers. However, such a vulnerability does not guarantee exploitation, as the access controls actively block unauthenticated access to vulnerable functions. To date, it is still unknown how well access controls are configured in existing smart contacts, and how many existing vulnerabilities are protected by access controls.

Developing access control techniques in smart contracts will involve an empirical study on the use and misuse of access controls. Existing `Ownable` access control is effective in restricting the access to the self-destroy function in smart contracts, but cannot handle complex access control models.

One future work direction is to develop access control techniques to support complex access control models with additional capabilities to configure the privilege of users. This will allow smart contracts to provide accurate protection to their critical fields and serve the functionality despite the adversarial actions.

# Bibliography


[1] Ethereum in bigquery: a public dataset for smart contract analytics, 2018. URL https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics. → pages 1, 29, 42

[2] Ether market capitalization chart, 2022. URL https://etherscan.io/chart/marketcap. → pages 1, 29, 42

[3] Swc-101 integer overflow and underflow, 2022. URL https://swcregistry.io/docs/SWC-101. → page 13

[4] Swc-104 unchecked return value, 2022. URL https://swcregistry.io/docs/SWC-104. → pages 3, 13

[5] Swc-107 reentrancy, 2022. URL https://swcregistry.io/docs/SWC-107. → page 13

[6] Swc-128 dos with block gas limit, 2022. URL https://swcregistry.io/docs/SWC-128. → page 13

[7] E. Albert, S. Grossman, N. Rinetzky, C. Rodríguez-Núñez, A. Rubio, and M. Sagiv. Taming callbacks for smart contract modularity. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. → page 20

[8] L. Alt and C. Reitwiessner. Smt-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 376–388. Springer, 2018. → page 27

[9] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura. Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 47–59, 2021. → page 18

[10] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017. → pages 2, 3, 13, 14, 16

[11] K. Avijit, P. Gupta, and D. Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows. *Software: Practice and Experience*, 36(9):971–998, 2006. → page 21

[12] G. Ayoade, E. Bauman, L. Khan, and K. Hamlen. Smart contract defense through bytecode rewriting. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 384–389. IEEE, 2019. → page 27

[13] S. M. Beillahi, E. Keilty, K. Nelaturu, A. Veneris, and F. Long. Automated auditing of price gouging tod vulnerabilities in smart contracts. → page 17

[14] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The keccak sha-3 submission, sha-3 competition (round 3), 2011. URL https://keccak.team/files/Keccak-submission-3.pdf. → page 51

[15] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016. → page 20

[16] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. *arXiv preprint arXiv:2104.08638*, 2021. → page 17

[17] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. *URL: https://ethereum.org/en/whitepaper/*, 2014. → page 1

[18] V. Buterin. Eip-150: Gas cost changes for io-heavy operations, 2016. URL https://eips.ethereum.org/EIPS/eip-150. → page 11

[19] C. Cadar and A. F. Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 765–768, 2016. → page 18

[20] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. DEFECTCHECKER: Automated smart contract defect detection by analyzing EVM bytecode. *IEEE Transactions on Software Engineering*,

pages 1–1, 2021. doi:10.1109/tse.2021.3054928. URL https://doi.org/10.1109/tse.2021.3054928. → page 17

[21] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng. Understanding code reuse in smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 470–479. IEEE, 2021. → pages 6, 29

[22] Y. Chen and C. Bellavitis. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, 13:e00151, 2020. → page 1

[23] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura. Ra: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 327–336. IEEE, 2020. → page 18

[24] ConsenSys. Mythril github repository, 2018. URL https://github.com/ConsenSys/mythril. → pages 7, 13, 18, 19

[25] G. Danezis and S. Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015. → page 1

[26] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020. → pages 19, 22

[27] P. Dutta, T.-M. Choi, S. Somani, and R. Butala. Blockchain technology in supply chain operations: Applications, challenges and research opportunities. *Transportation research part e: Logistics and transportation review*, 142:102067, 2020. → page 1

[28] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019. → pages 4, 17, 18, 19, 21

[29] Y. Feng, E. Torlak, and R. Bodik. Summary-based symbolic evaluation for smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1141–1152, 2020. → page 18

[30] J. Frank, C. Aschermann, and T. Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774. USENIX Association, Aug. 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/frank. → page 18

[31] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen. Easyflow: Keep ethereum away from overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 23–26. IEEE, 2019. → page 27

[32] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020. → pages 4, 18, 22

[33] A. Ghaleb, J. Rubin, and K. Pattabiraman. etainter: Detecting gas-related vulnerabilities in smart contracts. 2022. → page 17

[34] J.-R. Giesen, S. Andreina, M. Rodler, G. O. Karame, and L. Davi. Practical mitigation of smart contract bugs. *arXiv preprint arXiv:2203.00364*, 2022. → page 21

[35] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018. → pages 17, 21

[36] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020. → page 19

[37] A. Groce, I. Ahmed, J. Feist, G. Grieco, J. Gesi, M. Meidani, and Q. Chen. Evaluating and improving static analysis tools via differential mutation analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 207–218. IEEE, 2021. → pages 4, 19, 22

[38] K. Hara, T. Takahashi, M. Ishimaki, and K. Omote. Machine-learning approach using solidity bytecode for smart-contract honeypot detection in the ethereum. In *2021 IEEE 21st International Conference on Software*

*Quality, Reliability and Security Companion (QRS-C)*, pages 652–659. IEEE, 2021. → page 18

[39] A. Hefele, U. Gallersdörfer, and F. Matthes. Library usage detection in ethereum smart contracts. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 310–317. Springer, 2019. → page 27

[40] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert. The immutability concept of blockchains and benefits of early standardization. In *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, pages 1–8. IEEE, 2017. → page 47

[41] A. Holkner and J. Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28, 2009. → page 21

[42] S. Hwang and S. Ryu. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 542–553, 2020. → pages 5, 16, 22

[43] S. Jeon, G. Lee, H. Kim, and S. S. Woo. Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert. 2021. → page 18

[44] S. Ji, J. Dong, J. Qiu, B. Gu, Y. Wang, and T. Wang. Increasing fuzz testing coverage for smart contracts with dynamic taint analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 243–247. IEEE, 2021. → page 19

[45] B. Jiang, Y. Liu, and W. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018. → page 19

[46] E. Lai and W. Luo. Static analysis of integer overflow of smart contracts in ethereum. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, pages 110–115, 2020. → page 27

[47] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020. → page 20

[48] Z. Liao, S. Song, H. Zhu, X. Luo, Z. He, R. Jiang, T. Chen, J. Chen, T. Zhang, and X.-s. Zhang. Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts. *IEEE Transactions on Software Engineering*, 2022. → page 68

[49] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018. → page 19

[50] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282*, 2021. → page 18

[51] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021. doi:10.1109/TKDE.2021.3095196. → page 18

[52] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *Asian Symposium on Programming Languages and Systems*, pages 139–160. Springer, 2005. → page 21

[53] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016. → pages 4, 7, 17, 18, 19, 21

[54] A. Mavridou and A. Laszka. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In *International conference on principles of security and trust*, pages 270–277. Springer, 2018. → page 20

[55] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey. Verisolid: Correct-by-design smart contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019. → page 20

[56] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019. → page 4

[57] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019. → page 19

[58] S. Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin.org/bitcoin.pdf*, 2008. → pages 1, 10

[59] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020. → page 19

[60] T. D. Nguyen, L. H. Pham, and J. Sun. Sguard: Towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1215–1229. IEEE, 2021. → page 21

[61] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, 2018. → page 19

[62] openzeppelin. Proxy patterns, 2018. URL https://blog.openzeppelin.com/proxy-patterns/. → page 47

[63] OpenZeppelin. Openzeppelin safemath library, 2022. URL https://docs.openzeppelin.com/contracts/4.x/api/utils. → page 13

[64] S. Palladino. The parity wallet hack explained, 2017. URL https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7. → page 4

[65] Z. Pan, T. Hu, C. Qian, and B. Li. Redefender: A tool for detecting reentrancy vulnerabilities in smart contracts effectively. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 915–925. IEEE, 2021. → page 19

[66] Paradigm. Foundry, 2021. URL https://github.com/foundry-rs/foundry. → page 19

[67] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM joint*

*meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 912–915, 2018. → page 20

[68] C. Peng, S. Akca, and A. Rajan. Sif: A framework for solidity contract instrumentation and analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–473. IEEE, 2019. → page 39

[69] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020. → page 20

[70] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz. On blockchain and its integration with iot. challenges and opportunities. *Future generation computer systems*, 88:173–190, 2018. → page 1

[71] M. Ribeiro, P. Adão, and P. Mateus. Formal verification of ethereum smart contracts using isabelle/hol. In *Logic, Language, and Security*, pages 71–97. Springer, 2020. → page 20

[72] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2010. → page 21

[73] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, pages 52–78. Springer, 2011. → page 21

[74] M. Rodler, W. Li, G. O. Karame, and L. Davi. Evmpatch: timely and automated patching of ethereum smart contracts. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021. → pages 4, 5, 20, 63

[75] S. So, S. Hong, and H. Oh. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378, 2021. → pages 4, 18

[76] S. Srikant. *Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations*. PhD thesis, Massachusetts Institute of Technology, 2020. → page 18

[77] T. Sun and W. Yu. A formal verification framework for security issues of blockchain smart contracts. *Electronics*, 9(2):255, 2020. → page 20

[78] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020. → page 68

[79] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng. Soltype: refinement types for arithmetic overflow in solidity. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022. → page 17

[80] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018. → pages 17, 18, 19, 21

[81] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[82] C. F. Torres, M. Steichen, et al. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th {USENIX} security symposium ({USENIX} security 19)*, pages 1591–1607, 2019. → page 19

[83] C. F. Torres, H. Jonker, and R. State. Elysium: Automagically healing vulnerable smart contracts using context-aware patching. *arXiv preprint arXiv:2108.10071*, 2021. → page 20

[84] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018. → pages 4, 7, 17, 18, 19, 21

[85] F. Vogelsteller and V. Buterin. Eip-20: Token standard, 2015. URL https://eips.ethereum.org/EIPS/eip-20. → page 47

[86] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang. Smart contract security: a practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1410–1422. IEEE, 2021. → pages 4, 5, 6, 29

[87] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. Contractward: Automated vulnerability detection models for ethereum smart contracts.

*IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2020. → page 18

[88] Z. Wang, X. Chen, X. Zhou, Y. Huang, Z. Zheng, and J. Wu. An empirical study of solidity language features. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 698–707. IEEE, 2021. → pages 5, 16, 22

[89] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE*, pages 378–389, 2021. → pages 18, 21

[90] R. Xi and K. Pattabiraman. When they go low: Automated replacement of low-level functions in ethereum smart contracts. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022. → page 22

[91] R. Xi and K. Pattabiraman. Gohigh's dataset, Jan. 2022. URL https://doi.org/10.5281/zenodo.5843540. → page 42

[92] C. Xing, Z. Chen, L. Chen, X. Guo, Z. Zheng, and J. Li. A new scheme of vulnerability analysis in smart contract with machine learning. *Wireless Networks*, pages 1–10, 2020. → page 18

[93] Y. Xu, G. Hu, L. You, and C. Cao. A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks*, 2021, 2021. → page 18

[94] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1029–1040. IEEE, 2020. → pages 17, 21

[95] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017. → page 68

[96] R. Yu, J. Shu, D. Yan, and X. Jia. Redetect: Reentrancy vulnerability detection in smart contracts with high accuracy. In *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, pages 412–419. IEEE, 2021. → pages 19, 63

[97] D. A. Zetzsche, D. W. Arner, and R. P. Buckley. Decentralized finance. *Journal of Financial Regulation*, 6(2):172–203, 2020. → page 1

[98] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 23–34. IEEE, 2020. → pages 20, 64