12-2023

Towards securing smart contracts systematically

Duy Tai NGUYEN

Singapore Management University, dtnguyen.2019@phdcs.smu.edu.sg

Towards Securing Smart Contracts Systematically

TAI D. Nguyen

SINGAPORE MANAGEMENT UNIVERSITY

2023

Towards Securing Smart Contracts Systematically

TAI D. Nguyen

Submitted to School of Computing & Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

SINGAPORE MANAGEMENT UNIVERSITY
2023

Declaration of Authorship

I hereby declare that this dissertation is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in this dissertation.
This dissertation has also not been submitted for any degree in any university previously.

Signed: [illegible]

Date: 28/12/2023

Abstract

Smart contracts are a groundbreaking technique that allows users to programmatically modify the state of the blockchain. They are essentially self-enforcing programs that are deployed and executed on top of the blockchain. In recent years, we have witnessed various smart contract incidents that led to substantial financial losses and even business closures. These incidents mainly arise from design flaws in Solidity, a dominant programming language for writing smart contracts, which complicates the process of detecting and repairing vulnerabilities. Furthermore, there is a growing interest in attacking smart contracts by the attackers. This thesis is dedicated to developing effective methods to ensure the safety and correctness of smart contracts systematically. Our methods have two parts: *vulnerability detection* and *smart contract repair*. While the goal of vulnerability detection is to aggressively uncover bugs, smart contract repair eliminates detected bugs by adding safety constraints.

In the first part of the thesis, we primarily concentrate on vulnerability detection. We start by building a grey-box fuzzing engine for detecting common vulnerabilities like reentrancy and arithmetic vulnerabilities. The main contribution is an algorithm, inspired by search-based software testing (SBST), to improve the quality of the test suite. Subsequently, we design a formal verification framework to guarantee the correctness of smart contracts. The framework provides an expressive verification language and a functional verification engine that aims to eliminate global analysis and reduce false positives.

In the second part of the thesis, we propose repair algorithms to systematically eliminate detected vulnerabilities in smart contracts. We first design a novel approach to patch vulnerable implementations by analyzing control and data dependencies in their bytecode. Each vulnerability is defined in the form of dependencies and is patched using the corresponding templates. The patched contracts are proven to be free of vulnerabilities and incur low gas overhead. After that, we develop an algorithm to repair bugs in user-developed specifications in the form of a precondition/post-condition for each function. The algorithm is inspired by abductive inference and constraint-solving. It first automatically discovers inconsistencies between the specification and the implementation and then generates recommendations for repairing specifications. With vulnerability detection and contract repair, this thesis paves the way for achieving smart contract security systematically.

Contents

List of Figures

List of Tables

Acknowledgements

Four years of pursuing a Ph.D. is a long journey. I have experienced both the most rewarding and challenging moments of my life. I have been lucky to meet and work with talented people and have greatly appreciated their support and encouragement.

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor SUN Jun, not only for his guidance and support, but also for giving me an opportunity to explore my research interests. He guided me through the very first day when I was completely inexperienced in research. As of now, I have some achievements. Thank you for being patient with me during the time I was deeply demotivated.

I would like to thank my senior, Dr. Pham Hong Long, who worked closely with me. He acted as my secondary supervisor, gave me invaluable advice, and encouraged me to make me a better researcher.

I am also very grateful to the members of the defense committee, Professor David Lo, Associate Professor JIANG Lingxiao, and Assistant Professor Yi Li, for their valuable time and effort spent on reviewing my thesis and for providing me with insightful feedback. I also want to thank my collaborators, Tran Minh Quang, Yun Lin, Le Quang Loc, and Fu Song for their contributions to discussions, writing efforts, and valuable suggestions.

It would not have been possible without sharing the best moments with my friends at SMU and SUTD: Lyly Tran, Minh Nguyen, Sinh Huynh, Quang Pham, James, Vu Tran, Thu Tran, Kien Nguyen, Phuc Nguyen, Hoang Le, Hieu Do, Viet Tran, Gao Bo, Shi Ling, Becky, Raven. I want to especially thank Minh Nguyen for various topic discussions during the COVID pandemic. I had the delightful experience of hiking and solving puzzles with Quang Pham, and playing snooker with Tuan Tran and Kien Nguyen.

Last but not least, I would like to thank my family, especially my mom and dad for their endless support. I am deeply thankful to Giang Tran. Thanks for being with me during the toughest time, loving me, and being so patient with me. The last person is my beloved, Cam. You are the greatest motivation for me to become better.

Chapter 1

Introduction

In this chapter, we first introduce the concepts of blockchain and smart contracts. Subsequently, we present our research questions and their corresponding solutions. Lastly, we summarize the contributions of this thesis.

1.1 Smart Contracts

On September 15th, 2008, the fourth-largest investment bank Lehman Brothers filed for bankruptcy. It was the largest bankruptcy in U.S. history at that time with 25,000 employees worldwide affected. The collapse cost an estimated $10 trillion in lost economic output. It was among the most significant events of the financial crisis of 2007–08. After the collapse, many people started to question the trustworthiness of the centralized financial institutions. They sought for alternatives that offer them transparency.

In 2008, an anonymous person/group known as Satoshi Nakamoto created Bitcoin, a fully decentralized cash system. This makes Bitcoin the first cryptocurrency to solve the double-spending problem [1] without the need for a third party such as governments or financial institutions. Bitcoin operates on

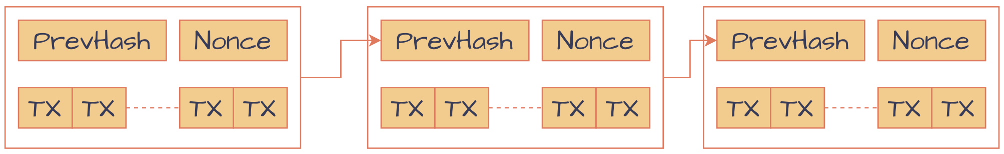


FIGURE 1.1: An example of a *blockchain*

a peer-to-peer network and ensures the transparency by cryptographic techniques. The underlying technology of Bitcoin is *blockchain*. Initially, Bitcoin had no value and was not traded. In 2010, Laszlo Hanyecz made the first real-world purchase – swapping 10,000 Bitcoin for two pizzas [2]. This remarkable event is celebrated annually as "Bitcoin Pizza Day". After that, Bitcoin continued to gain popularity and recognition, it hit its all-time high of over $68,500 in 2021. Nowadays, Bitcoin becomes the most valuable digital asset with a daily trading volume of more than 21 billion USD and a market capitalization worth almost 569 billion USD [3].

A blockchain is a shared, immutable, and decentralized ledger that stores all transactions conducted in a peer-to-peer network. It contains a growing list of records, known as blocks, that are securely linked together via cryptographic hashes [4] as shown in Figure 1.1. A block typically consists of transactions, a block link, and a nonce. It is connected to the ones before and after it to form a chain of blocks. A block link is a cryptographic hash of the previous block. It prevents any block from being altered, tampered or inserted between two existing blocks. A nonce is a random number that can be used in conjunction with data in the block to create a unique block link. Once a blockchain is formed, it is immutable and irreversible.

Bitcoin is powered by blockchain technology. However, it is primarily restricted to decentralized payments. A pioneering breakthrough occurred in 2013 when Vitalik Buterin introduced Ethereum and demonstrated that Ethereum

```
pragma solidity 0.4.24;

contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to) payable public{
    credit[to] += msg.value;
  }
  function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
  }
  function queryCredit(address to) view public returns(uint){
    return credit[to];
  }
}
```

FIGURE 1.2: A simple smart contract excerpted from SWC [6]

could be used for multiple purposes e.g., real estate, supply chains, and insurance. Ethereum is different from Bitcoin in various aspects. However, the primary aspect that sets them apart is the ability to execute so-called *smart contracts*. Smart contracts are essentially self-enforcing programs that are deployed on Ethereum blockchain and executed within Ethereum runtime environment, namely Ethereum Virtual Machine (EVM). An EVM is a stack-based machine that operates on a large set of instructions [5] to compute the state of the blockchain.

Despite a large variety of programming languages (e.g., Solidity [7], Vyper [8], and Bamboo [9]), Solidity is the most dominant language for implementing smart contracts in Ethereum. It is a statically-typed curly-braces programming language. A Solidity smart contract is similar to a class in object-oriented programming languages such as Java or C#. It contains persistent data such as state variables and functions that can modify these variables. Figure 1.2 shows a simple smart contract implemented in Solidity. The contract `SimpleDAO` has 3 public functions including `donate`, `withdraw`, and `queryCredit`. It works as a bank service where a user can call the function `donate` to transfer their ethers (i.e., the native cryptocurrency of Ethereum) to another user (lines 6-8), call the function `withdraw` to get their own ethers back (lines 10-15), and call the function `queryCredit` to query the deposited balance of any user (lines 17-19). Several global features are used including variables `msg.value` (i.e., the number of wei [1] sent by the caller), `msg.sender` (i.e., the address of the caller), and function `<receiver>.call.value` (i.e., transferring ethers from the caller to the receiver). With the capabilities of smart contracts, Ethereum has risen to become the second-largest cryptocurrency in the world in terms of market capitalization, standing just right behind Bitcoin [10]. Thus, any smart contract mistakes could potentially result in huge financial losses.

1.2 Vulnerabilities

Like traditional programs, smart contracts are subjected to code-based vulnerabilities. Unlike traditional programs, they are immutable, meaning that smart contracts cannot be amended once deployed. Any vulnerabilities on Ethereum become permanent and cannot be rectified. There have been various smart contracts attacks in the past. We highlight three incidents that caused great financial loss and resulted in controversial decisions from Ethereum Foundation.

The DAO is a decentralized autonomous organization that allows participants to vote for various proposals and decisions. All votes and activities are managed by smart contracts and posted on Ethereum. The DAO launched in late April 2016 and raised more than $150 million after a month of selling tokens. By May 2016, it held around 14% of the issued ether, which is the largest

[1] 1 ether = 1e18 wei

crowdfunding of all time. In June 2016, The DAO [11] was subjected to a so-called *reentrancy* attack. The attackers gained access to 3.6M ethers (≈$50M). To recover the funds, Ethereum Foundation made a controversial decision [12]–[14]. That is, they released a hard fork to revert the attack. This led to the creation of two separate blockchains: Ethereum and Ethereum Classic.

A multi-signature (multi-sig) wallet is a set of contracts that manages ether. It requires two or more private keys to move funds out of the wallet. In July 2017, an attacker exploited an insecure *delegatecall* statement to steal 150k ethers (≈$30M) from the Parity wallet in two transactions. In the first transaction [15], the attacker called a flawed constructor `initWallet` to obtain ownership of the wallet. In the second transaction [16], the function `execute` was invoked to transfer all funds to the attacker's account.

A few months later, the Parity wallet suffered from the second attack. This time, an attacker *devops199* executed an unrestricted `selfdestruct` statement to destroy access to 514k ethers (≈$267M). First, the attacker called the function `initWallet` to become the owner [17]. Then, the attacker attempted to drain funds by executing the function `kill` [18]. However, all funds were locked. Approximately 30 minutes later, the attacker created a Github issue #6995 [19] and announced the failed attack.

1.3 Problem Definition

These aforementioned incidents raised serious concerns regarding the security problems of smart contracts. Furthermore, many new smart contract-specific vulnerabilities have been discovered and exploited during the development of Ethereum [20]–[22]. This motivates us to develop tools and theories to ensure the safety and correctness of smart contracts systematically. The overarching problem is defined as follows.

Problem statement: *Given a smart contract and its specifications, our objective is to develop automated tools that aim to ensure that the given smart contract satisfies its specifications.*

To tackle the problem, a well-known approach is to reduce or even eliminate vulnerabilities present in smart contracts. There exist many solutions to the above problem, such as Oyente [23], Securify [24], and ILF [25]. We constraint the solution space by focusing on two main research directions: *vulnerability detection* and *program repair*.

- **Vulnerability detection**: aims to aggressively uncover bugs. Vulnerability detection tools are developed to search for concrete inputs that violate specifications. It is noted that the specifications could be a set of generic requirements, such as free of overflow and free of re-entrancy or manually specified requirements.
- **Program repair**: aims to eliminate all detected bugs without breaking the functionality of the smart contract. Program repair tools are designed to modify a contract or its specifications to ensure that the given contract satisfies its specifications. It is noted that specifications can be modified by assuming the implementation is correct.

1.4 Research Questions

For each of the two approaches that we focus on, we first conduct a literature review to gain a comprehensive understanding of the existing approaches. After that, we formulate research questions to address the limitations of the existing approaches or explore new ideas. By providing well-founded answers to the research questions, we make meaningful contributions to the respective research domain.

A literature review at the time of the study shows that there are a few works such as Oyente [23], ContractFuzzer [26], teEther [27], MAIAN [28], Osiris [29] that aim to discover vulnerabilities. Oyente [23] is the first symbolic execution tool. It analyzes bytecode to identify four different types of bugs including *re-entrancy*, *transaction ordering dependency*, *timestamp dependency*, and *missed handle exceptions*. Although Oyente is capable of finding thousands of bugs on Ethereum, it is neither sound nor complete. Subsequent works formulate vulnerabilities in many different ways and rely on these formulas to improve their detecting accuracy. In 2018, Krupp and Rossow presented teEther [27], which searches for certain critical paths (i.e., contain ether transfer) and generates exploits for them. The exploit is generated by solving symbolic constraints constructed from critical paths. In the same year, Nikolic et al proposed MAIAN [28]. MAIAN detects vulnerabilities across a long sequence of invocations of a contract. Its vulnerable contracts are categorized into *greedy* contracts (lock funds indefinitely), *prodigal* contracts (leak funds to arbitrary users), and *suicidal* contracts (be susceptible to be killed by any user). Later on, Torres et al. presented Osiris [29], which combines symbolic execution and taint analysis to discover various types of integer bugs. Among them, ContractFuzzer [26] is the only fuzzing work. Although it is able to detect seven kinds of vulnerabilities by injecting code into EVM to analyze behaviors of smart contracts regardless of the inputs, it does not incorporate feedback to improve the quality of the test suite, which is considered the most important part of a fuzzer. This leads to our first research question.

Research Question 1 (RQ1): *How to develop a fuzzing engine that can improve the quality of the test suite, increase code coverage, and discover more vulnerabilities?*

Existing approaches for detecting vulnerabilities of smart contracts focus on a collection of generic bugs (e.g., reentrancy, overflow or underflow, frontrunning, and frozen funds). While these approaches are undoubtedly useful, they are incapable of pinpointing contract-specific bugs or showing their absence. Built upon the idea that different contract has different correctness specification, several recent approaches, such as VerX [30], SmartPulse [31], Solc-verify [32] have been developed to support the falsification or verification of manually specified correctness specifications. VerX [30] focuses on temporal properties of Ethereum contracts. It reduces the temporal safety verification to reachability verification and applies the state-of-the-art reachability checking. SmartPulse [31] relies on counter example-guided abstraction refinement algorithm (CEGAR), but exploits domain-specific knowledge about smart contracts to make verification more efficient. Solc-verify [32] translates Solidity contracts into the Boogie intermediate language and relies on the Boogie system for verification. It supports contract invariant, loop invariant, and pre-/post conditions. Although the above-mentioned verification works are useful, they suffer from shortcomings such as a specification language with limited expressiveness and high false alarm rates. This leads to our second research question.

Research Question 2 (RQ2) *How to build a compositional verifier that supports a user-friendly and expressive specification language and ensures the correctness of functional specifications?*

While vulnerability detection is essential for identifying security weaknesses, it does not remove the vulnerabilities. Program repair is a complementary approach, which mitigates the detected vulnerabilities. Existing works such as SmartShield [33], Elysium [34], and EVM-patch [35] patch vulnerable smart contracts in different ways. Elysium [34] uses an outsourced bug localization to determine vulnerable code. Then, it infers context information from bytecode to generate a template-based patch. EVMPatch [35] features a bytecode rewriting engine and leverages a template-based approach to patch integer overflow and access control bugs. SmartShield [33] extracts control and data dependencies from bytecode and source-code, then uses extracted semantics to fix insecure control flows and data operations. Although the mentioned works are able to repair vulnerable smart contracts, none of them guarantee the safety and correctness of patched smart contracts. This led to our third research question.

Research Question 3 (RQ3) *How to patch a smart contract or its specifications in a way that the patched contract is guaranteed to satisfy its specifications?*

1.5 Contributions

To effectively tackle the aforementioned research questions, this thesis makes several contributions. First, we propose four different approaches, varying from dynamic analysis to static analysis, to ensure the safety and correctness of smart contracts. Second, we implement our approaches as self-containing tools including sFuzz, sGuard, iContract, SPAIR and make them publicly available [36]. Lastly, for each tool, we thoroughly evaluate its efficiency through various experiments and report detailed results. The main contribution of each work is summarized as follows.

We develop sFuzz, a feedback-guided fuzzing engine for smart contracts running on Ethereum. It is inspired by AFL [37], a well-known fuzzer for C programs. Although AFL-based fuzzing is often effective, it has limitations as well. For example, AFL is often expensive in covering branches guarded with strict conditions. To tackle the problem, sFuzz integrates AFL-based fuzzing with an efficient lightweight adaptive strategy for selecting seeds. Although inspired by search-based software testing [38], [39], the latter distinguishes itself by having a lightweight objective function (designed considering characteristics of Solidity programs) as well as a novel multi-objective optimization strategy. sFuzz has been applied to more than 4 thousand smart contracts and the experimental results show that (1) sFuzz is efficient, e.g., two orders of magnitude faster than state-of-the-art tools; (2) sFuzz is effective in achieving high code coverage and discovering vulnerabilities; and (3) the different fuzzing strategies in sFuzz complement each other.

We present iContract, a fully compositional verification system for verifying and enforcing the correctness of smart contracts. Existing approaches, such as Solc-verify [32], VerX [30], SmartPulse [31], may have less expressive specification language, high false alarms, and do not provide a way to enforce the specification. Therefore, we design a static verification system with a specification language that supports fully compositional verification with the help of function invariants, contract invariants, loop invariants, and call invariants. Our approach automatically proves the correctness of a smart contract statically or checks the unverified part of the specification during runtime. Using iContract, we have verified 10 high-profile smart contracts against manually developed detailed specifications, many of which are beyond the capacity of existing verifiers. In particular, we have uncovered two ERC20 violations in the BNB and QNT contracts.

We propose sGuard, which automatically fixes potentially vulnerable smart contracts. It is inspired by program fixing techniques for traditional programs such as C or Java. Existing fixing approaches, such as GenFrog [40], PAR [41], Sapfix [42], often suffer from the problem of weak specifications, i.e., a test suite is taken as the correctness specification. A fix driven by such weak correctness criteria may over-fit the given test suites and does not provide a correctness guarantee in all cases. Furthermore, fixes for smart contracts may suffer from not only time overhead but also gas overhead (i.e., extra fees for running the additional code). Therefore, we develop an approach that automatically transforms smart contracts so that they are provably free of four common kinds of vulnerabilities. The key idea is to apply run-time verification in an efficient and provably correct manner. Experiment results with 5000 smart contracts show that our approach incurs minor run-time overhead in terms of time (i.e., 14.79%) and gas (i.e., 0.79%).

We propose SPAIR, the first approach that automatically discovers inconsistency between the specifications and the implementation, and generates recommendations for repairing the specifications. Given a smart contract with potentially buggy specifications (in the form of a precondition/postcondition for each function), SPAIR works in three steps. Initially, it creates a call graph from the smart contract, in which each node is a function and there is a directed edge from the caller to the callee. The graph is then divided into connected components. Subsequently, SPAIR patches the specifications in a bottom-up manner by identifying connected components containing functions associated with incorrect specifications. Lastly, SPAIR ranks all generated patches and selects the best one for each function. Experiments on 10 high-profile smart contracts show that SPAIR is able to recommend all the desired specifications. The repairing process is not limited to specifications that contain a single bug but also handles specifications with multiple bugs efficiently within a minute in most cases.

1.6 Outline

This thesis consists of two main parts. The first part includes two chapters that propose approaches to detect vulnerabilities and verify the correctness of smart contracts automatically. This part is based on two works [43], [44], which are joint works with *Long H. Pham, Jun Sun, Yun Lin, Quang Tran Minh, and Quang Loc Le*. The second part contains another two chapters that focus on repairing buggy implementations and specifications. This part is based on two works [45], [46], which are joint works with *Long H. Pham, Jun Sun, Yan Wan, and Fu Song*.

Each chapter focuses on one work, except Chapters 1-2 and Chapter 7. It aims to offer self-contained information. However, the readers are advised to first read the background in Chapter 2 before delving into the subsequent chapters. The detailed outline of this thesis is as follows.

Chapter 2: lays the foundation necessary for comprehending our works. This chapter presents Ethereum, a groundbreaking blockchain, as well as the fundamental concepts of smart contracts.

Part I: Smart Contract Vulnerability Detection

Chapter 3: addresses the RQ1 by presenting sFuzz that relies on branch coverage and branch distance to improve the quality of the test suite. sFuzz is able to detect nine distinct types of vulnerabilities in smart contracts. This chapter is based on the paper:

* Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., & Minh, Q. T. (2020, June). sFuzz: An efficient adaptive fuzzer for solidity smart contracts. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 778-788).*

Chapter 4: addresses the RQ2 by proposing iContract for verifying and enforcing the correctness of smart contracts. This chapter is based on the paper:

* Nguyen, T. D., Pham, L. H., Sun, J., & Quang, L. L. (2023, August). iContract: An Idealist's Approach for Smart Contract Correctness. *In Proceedings of The 24th International Conference on Formal Engineering Methods.*

Part II: Smart Contract Vulnerability Repair

Chapter 5: addresses the RQ3 by proposing sGuard that can identify vulnerabilities at the bytecode level and patch vulnerable code at the source code level. The patched code has low gas overhead. Moreover, it is proven to be free of four distinct types of vulnerabilities. This chapter is based on the paper:

* Nguyen, T. D., Pham, L. H., & Sun, J. (2021, May). sGuard: Towards Fixing Vulnerable Smart Contracts Automatically. *In 2021 IEEE Symposium on Security and Privacy (pp. 1215-1229).*

Chapter 6: addresses the RQ3 by proposing SPAIR that repairs buggy specifications. We assume that specifications written by humans can be buggy. Rather than repairing the implementation, as discussed in Chapter 5, we repair specifications using abductive inference and constraint solving. The results are the set of verified specifications. This chapter is based on the paper:

∗ Nguyen, T. D., Pham, L. H., Sun, J., Wan, Y., Song F. SPAIR: Towards Repairing Smart Contract Specification. *Submitted to ICSE in 2023 August*.

Chapter 7: concludes this thesis by summarizing results and discussing limitations and future works.

Chapter 2

Preliminaries

In this chapter, we provide the background knowledge necessary to understand the works in this thesis. We briefly introduce Ethereum, smart contracts, and their vulnerabilities.

2.1 Ethereum

Ethereum is a global, open-source blockchain technology for building decentralized applications (DApps). It is powered by smart contracts and embedded with a native digital currency, ether (ETH). It was created in 2015 by Vitalik Buterin to expand upon the primary function of Bitcoin. Figure 2.1 shows an Ethereum chain that is viewed as a transaction-based state machine. The machine executes all transactions in a block and transitions from the current state to a new state. Ethereum starts a genesis state and ends in the current state.

A wide range of applications can be built on top of Ethereum such as finance, games, and metaverse. Among them, decentralized finances are the most widely used applications. They focus on providing financial services using cryptocurrencies (e.g., token USDT [47], BNB [48]). Decentralized finances offer the likes of lending, borrowing, earning interest, and private payments.

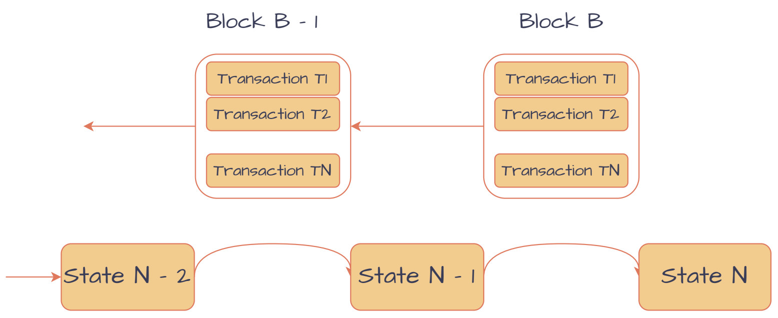


FIGURE 2.1: Ethereum can be viewed as a transaction-based state machine

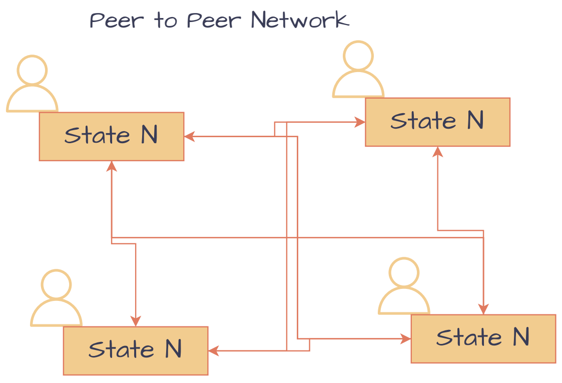


FIGURE 2.2: Decentralised nodes constitute Ethereum P2P network

For example, lending and borrowing protocols such as Aave [49] and Compound [50] allow participants to lend tokens and earn interest. Exchange protocols such as Uniswap [51], and Sushiswap [52] allow users to exchange a token for another token and pay exchange fees.

2.1.1 Consensus

Ethereum is built on top of a peer-to-peer network (see Figure 2.2) where participants (i.e., nodes) rely on a consensus protocol, named Proof-of-Work (PoW), to validate a new block and append it to the blockchain. PoW was introduced by Bitcoin's creator. It requires so-called miners to run a power-consuming hashing algorithm to find the nonce for a new block. Only new blocks with a valid nonce are added to the chain. The chain with the newly appended block is subsequently shared with other nodes to ensure consensus. The PoW requires a large amount of energy to keep the chain safe. As a node can propose new blocks, there exists a situation where many new blocks simultaneously link to a block. This creates a tree, which contains multiple valid chains. Only the branch of the tree with the most cumulative computational effort or difficulty is considered the official chain. This is known as *heaviest subtree* rule [53]. Blocks that are valid but not in the official chain are ommer blocks.

In 2022, Ethereum officially switched from PoW to Proof-of-Stake (PoS) to solve the energy problem. PoS shares the same goal with PoW. However, validators (i.e., miners in PoW) have to stake ether to have the ability to append the new block to the chain. A validator is randomly selected by the algorithm. If 2/3 validators agree on the new block, the selected validator is awarded [54]. Otherwise, the selected validator loses its entire stake.

Ethereum offers secure and decentralized transaction capabilities, but it faces scalability limitations. Ethereum can process around 15 transactions per second (TPS) [55], which is much lower than other competitors like Solana [56], which has 710,000 TPS. This limited TPS capability in Ethereum leads to longer transaction confirmation time. Moreover, the cost of each transaction increases significantly during the network congestion. For example, in December 2017, transactions generated by Cryptokitty [57] caused network congestion by accounting for 12% of all transactions and the gas fees were anywhere from $100 to $200 transactions.

Layer 2 (L2) is a collection of scaling solutions that aim to increase transaction throughput without sacrificing decentralization or security. This is achieved by processing the transactions off the layer-1 blockchain (e.g., Ethereum), while

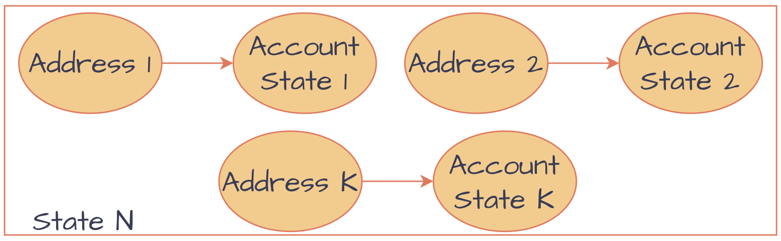


FIGURE 2.3: An account is a mapping between addresses and account states

still relying on its security. There are several types of L2, but the two most dominant types are:

- **Optimistic rollups**: such as Arbitrum One [58], which executes multiple transactions outside of Ethereum, bundle post-transaction data into a single batch, and submit them as a single transaction to Mainnet. They are *optimistic* in the sense that off-chain transactions are assumed to be valid and no proofs of validity are published.
- **Zero-knowledge rollups**: such as zkSync Era [59], which is similar to optimistic rollups, but they rely on a cryptographic technique called ZK-SNARK to publish the proofs of validity for off-chain transactions.

2.1.2 Accounts

An account is a mapping between addresses and account states (see Figure 2.3. It is mainly used to store balances and create transactions. In the Ethereum platform, there are two types of accounts:

- **Externally-owned accounts (EOAs)**: which are controlled by anyone with the corresponding private keys.
- **Contract accounts**: which are smart contracts deployed to the network, controlled by code.

Field	Description
Nonce	An incremental counter that indicates the number of transactions associated with this account. It is introduced to protect transactions from relay attacks where the signed transactions are duplicated and broadcasted by attackers.
Balance	The number of wei owned by this account. Wei is a denomination of Eth, i.e., 1 Eth = $1e18$ wei.
CodeHash	The hash of the EVM code of this account. This code is executed if this account receives a message call. For externally owned accounts, the CodeHash field is the hash of an empty string.
StorageRoot	A 256-bit hash of the root node of a Merkle Patricia trie that encodes the storage contents of the account. It is empty by default and is updated when data is written to the storage.

TABLE 2.1: Common fields of an Ethereum account

While both types of accounts have the ability to receive, hold, send ether, and interact with deployed smart contracts, they are different in some aspects. First, externally-owned accounts are created without any fees. In contrast, contract accounts require code execution and thus creating them has a cost. Second, externally-owned accounts can initiate transactions. Conversely, contract accounts can only respond to transactions. Last, transactions between externally-owned accounts do not trigger any code executions, whereas, transactions between smart contracts always trigger code execution. An account is identified by a 160-bit unique address. Its state consists of four different fields as shown in Table 2.1.

2.1.3 Transactions

Transactions are cryptographically signed instructions from EOAs to update the state of the Ethereum network. The sender of a transaction cannot be a contract. There are two types of transactions: *message calls* and *contract creations*.

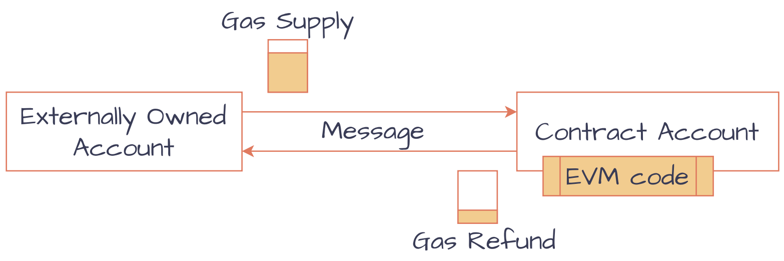


FIGURE 2.4: Gas is refunded if it is not used

Field	Description
From	A 160-bit address of the sender, which will be signing the transaction.
Recipient	A 160-bit address of the recipient, which will receive that transaction. If the recipient is a contract account, then its code is executed. Otherwise, the transaction transfers ether.
Signature	A 65-byte value is generated when the sender signs that transaction. It is used to verify the sender of the transaction.
Nonce	An incremental counter that indicates the number of transactions associated with the sender.
Value	The amount of ether to transfer from sender to recipient.
Input data	An optional field that includes arbitrary data. This is often used to store inputs for a message call.
GasLimit	A maximum amount of gas units that can be consumed by the transaction. This number is set before the transaction is processed.

TABLE 2.2: Common fields in an Ethereum transaction

A message call has a recipient (i.e., account address). A contract creation has an empty recipient. Table 2.2 shows common fields of a submitted transaction. It is noted that gas is a reference to the computational price required to process the transaction by a miner. To avoid denial-of-service (DoS) attacks, as well as reward miners, users have to pay a fee for this computation (see Figure 2.4).

TABLE 2.3: Common fields in an Ethereum block header

Field	Description
ParentHash	A 256-bit hash of the parent block's header
UncleHash	A 256-bit hash of the uncle block headers
beneficiary	A 160-bit address of the miner who mined the block
StateRoot	A 256-bit root hash of the Ethereum Merkle Patricia trie. It is computed after all transactions are executed
TransactionRoot	A 256-bit root hash of the transaction trie
ReceiptsRoot	A 256-bit root hash of the receipts trie
Difficulty	A scalar value corresponding to the difficulty level for mining
GasLimit	A scalar value that equal to the maximum gas allowed for transactions in the block
GasUsed	A scalar value that equal to the total gas used by all transactions in the block
Timestamp	A scalar value that equal to the unix timestamp of block creation
ExtraData	The additional data, often used for block metadata
MixHash	A 256-bit hash used in proof-of-work verification
Nonce	A 64-bit value which is the random number used in proof-of-work

2.1.4 Blocks

Blocks are batches of transactions with a link, which is the cryptographic hash of the previous block in the chain. Because the link contains the cryptographic hash, it prevents the chain of blocks from being tampered. That is, any change in a block would invalidate the links of all subsequent blocks. A block consists of a block header, together with information corresponding to the comprised transactions, and a set of block headers of ommers. The common fields in an Ethereum block header are shown in Table 2.3.

```
// SPDX-License-Identifier: MIT
pragma solidity ^ 0.8 .0;

contract Bank {
  mapping(address => uint) private balances;

  function deposit() public payable returns(uint) {
    balances[msg.sender] += msg.value;
    return balances[msg.sender];
  }

  function withdraw(uint withdrawAmount) public returns(uint) {
    if (withdrawAmount <= balances[msg.sender]) {
      balances[msg.sender] -= withdrawAmount;
      payable(msg.sender).transfer(withdrawAmount);
    }
    return balances[msg.sender];
  }
}
```

FIGURE 2.5: An example of a basic wallet.

2.2 Smart contracts

The concept of smart contracts was first proposed by Nick Szabo in 1997 [60]. It became a reality after the creation of Ethereum [61] in 2015. An Ethereum smart contract implements a set of rules that aim to manage digital assets in Ethereum accounts. Despite a large variety of contract programming languages (e.g., Solidity [62], Vyper [8], and Bamboo [9]), Solidity is the most dominant one for implementing smart contracts. In this section, we introduce Solidity and bytecode.

2.2.1 Solidity

Solidity is an object-oriented, statically typed, and high-level programming language. Each contract (i.e., class) can contain declarations of *state variables*, *functions*. State variables are variables whose values are permanently stored in contract storage. Functions contain a set of statements, which can be defined either inside or outside of contracts. There are two special functions: *constructor* and *fallback*. The constructor is used to initialize state variables when the contract is deployed. The fallback is executed if the function call does not match with any function in the contract. The entire structure of smart contracts is defined in the solidity website [62].

Solidity offers several elementary types such as *uintN, intN, address, bytes, bytesN* where N is the data size. These types can be used to form complex types such as *array, struct, mapping*. Among them, mapping is the most used type to keep track of balances. A mapping is a hashtable, where a key is associated with a value. It is stored in a storage and has no property to keep track of the number of elements. For example, `mapping(address=>unit) balances` is used to store the balance for a user, which is identified by an address. The expression `balances[msg.sender]` accesses the balance of the caller.

External function calls set Solidity apart from traditional programs. A call requires a receiver address, some ethers, and arguments to be invoked. It can transfer ether and execute remote code. External function calls facilitate the interaction between multiple smart contracts. With millions of deployed smart contracts, they may create complicated call chains that are error-prone. Table 2.4 shows a list of function calls where functions `send`, `transfer` are commonly used to transfer ether without code execution. The rest are low-level, powerful, and high-risk external calls.

Figure 2.5 shows an example of a basic bank implemented in Solidity. The contract declares a mapping (line 5) and two functions `deposit` (line 7) and `withdraw` (line 12). Participants invoke the function `deposit` to store ether and the function `withdraw` to get it back. It is noted that the check at line 13 is important to prevent potential underflow at line 14, where attackers try to

External call	Description
addr.send(y)	sends a given amount of ether y to address $addr$ and returns `false` on failure
addr.transfer(y)	sends a given amount of ether y to address $addr$ and throws on failure
addr.call(y)	executes the function located at address $addr$ and specified in y
addr.staticcall(y)	executes the function located at address $addr$ and specified in y. The called code is prohibited from modifying storage
addr.delegatecall(y)	executes the function located at address $addr$ and specified in y. The called function is executed within the calling context i.e., storage, balance of calling contract
selfdestruct(addr)	remove the called contract from the blockchain and clear its storage. The remaining ether is transferred to the address $addr$

TABLE 2.4: A list of Solidity-supported external function calls

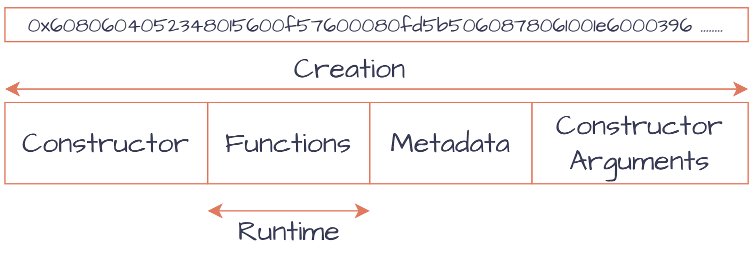

0x6080604052348015600f57600080fd5b5060878061001e6000396 ....
Creation
Constructor
Functions
Metadata
Constructor Arguments
Runtime

FIGURE 2.6: The layout of deployment bytecode

withdraw more than their deposited ether.

2.2.2 Bytecode

EVM bytecode is a sequence of bytes. Each byte is either an instruction or a single byte of data. There are two distinct types of bytecode: *creation* and *run-time* bytecode. The creation bytecode is used to create a new contract. The contract creation first creates a contract account, then executes the bytecode of the constructor, and last stores the run-time bytecode to the created contract account.

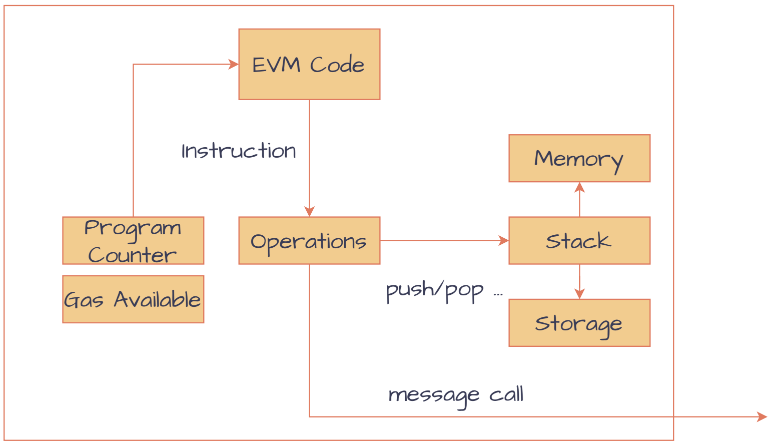


FIGURE 2.7: EVM architecture

While deployment bytecode is run once, run-time bytecode is executed whenever a transaction is sent to the contract.

Figure 2.6 shows the layout of a deployment bytecode. Besides run-time bytecode, the deployment bytecode also contains metadata and arguments of the constructor. Some compilers will append non-executable metadata to the end of run-time bytecode. This is to provide additional data to analysis tools or EVM.

2.2.3 Ethereum Virtual Machine

The Ethereum virtual machine (EVM) is a sandboxed run-time environment for running bytecode compiled from a high-level language such as Solidity and Vyper. It has a simple stack-based architecture, whose word size is 256 bits (i.e., size of stack items). The formal definition of the EVM is specified in the Ethereum Yellow Paper [61].

EVM supports more than 140 instructions. An instruction is represented by a one-byte opcode and its operands. For example, `0x01` is `ADD`. This opcode removes two items from the top of the stack and pushes the result i.e., the sum of two stack items. The number of operands is defined by the instruction. It operates on a stack and can access data on memory (i.e., volatile storage) and storage (i.e., non-volatile storage). Each instruction is associated with a gas cost. This cost is either dynamic or fixed. For example, an `ADD` instruction costs 3 gas. In contrast, `SSTORE` instruction costs 20,000 gas to store data in unallocated storage and costs 5000 to update allocated storage. It is important to understand how gas cost is computed to build gas-friendly contracts.

A stack is a temporary data store that is mainly used to store arguments and results of instructions during contract execution. It is a fixed array of 1024 items. A contract can add, remove, and change the order of items on the stack using instructions like `PUSH` (push data on top of the stack), `POP` (pop the top item on the stack), `SWAP1` (swap the top item with the second item on the stack). A stack operation is inexpensive in terms of gas costs.

A memory is a data store that is mainly used to handle complex data structures during contract execution. It is a dynamic array of 32 bytes and is initialized with zeros. A contract can read or write data to any location in the memory using instructions like `MLOAD` (load data) and `MSTORE` (store data). A memory operation is typically inexpensive in terms of gas costs compared to a storage operation. In general, memory could be divided into two regions: *reverse* and *data*. The reserve region is the place to store data for specific tasks such as storing the highest address of allocated data. The data region is the place for storing data for complex data variables such as an array of unsigned integers or a string.

A storage is a persistent data store that is used to store state variables. It is a hashing table of 32 bytes and is initialized with zeros. A contract can read or write data to any location in the storage using instructions like `SLOAD` (load

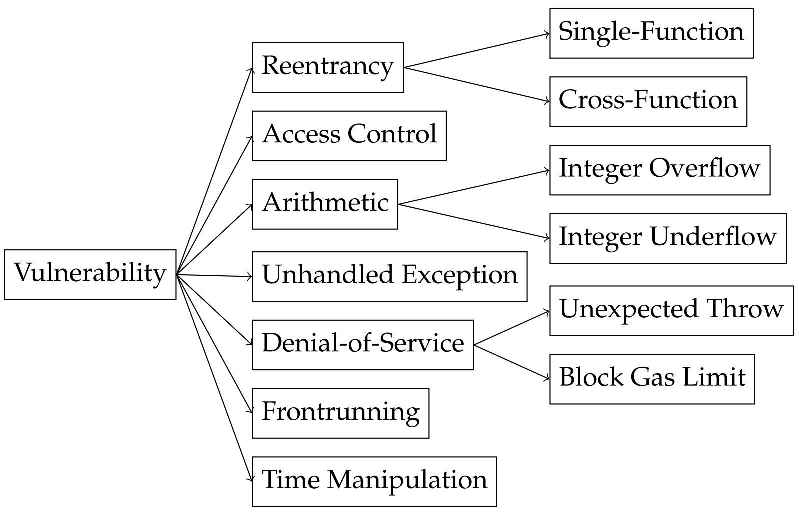


FIGURE 2.8: A taxonomy of vulnerabilities

data) and `SSTORE` (store data). A storage operation is expensive and typically is associated with dynamic gas cost.

2.3 Vulnerabilities

Like traditional programs, smart contracts are subjected to code-based vulnerabilities. Unlike traditional programs, they are immutable, meaning that smart contracts can not be amended once deployed. Any vulnerabilities on Ethereum become permanent and can not be rectified. As many infamous attacks occurred throughout the existence of Ethereum, Consensys [63] released a list of well-known attacks. This intends to provide a baseline knowledge of security considerations. However, the list is short. In 2019, a comprehensive list of vulnerabilities so-called *Smart Contract Weakness Classification Registry (SWC)* was published. SWC assigns an ID to a type of vulnerability and provides a description, as well as, sample codes. Since SWC aims to cover a full list of vulnerabilities, it contains many syntax-related and duplicated entries. In the following, we define eight different types of vulnerabilities as shown in Figure 2.8. It is noted that real-world vulnerabilities are not limited to our taxonomy, we attempt to provide a list of common and well-studied vulnerabilities.

Reentrancy: A reentrancy attack occurs when a function makes an external call to another function, and the called function calls back the original contract in the same transaction before finishing the original invocation. This vulnerability allows attackers to control the execution flow and illegally manipulate the storage state of the original contract. Many infamous incidents in the past are the result of exploiting reentrancy vulnerabilities. In June 2016, the notorious *The DAO* reentrancy attack led to a financial loss of $50M. To reverse the attack and return the stolen funds to the rightful owners but advocate the motto "code is law", the Ethereum Foundation was forced to perform a hard fork. This led to the creation of two separate Ethereum chains.

Reentrancy vulnerability can be categorized into two distinct types in terms of attacking flow: *single-function* and *cross-function* reentrancy. A single-function reentrancy attack targets a single vulnerable function, while a cross-function reentrancy attack targets multiple vulnerable functions. Besides the two main categories, the advanced work on reentrancy detection by Rodler et al. [64] defines additional types such as *delegated* reentrancy (i.e., related to `DELEGATECALL` instruction) and *created-based* reentrancy (i.e., related to `CREATE` instruction) to formulate a better detection algorithm. Despite the fact that reentrancy is a well-studied vulnerability, its detection is challenging as it requires multiple interactions between vulnerable contracts and attackers before the vulnerability is detected. To protect smart contracts from reentrancy attacks, developers are recommended to use methods `send` or `transfer` instead of `call`. Moreover, a mutex [65] could be added to prevent successive calls to the same contracts.

Access control: An access control attack occurs when attackers execute the code that is designed to be executed exclusively by authorized users. This kind of vulnerability occurs mainly due to the lack of standard access control. Developers implement access control in an ad-hoc manner, potentially leading to inconsistencies and vulnerabilities in smart contracts. For example, `tx.origin` returns the address of the person that originates the transaction. However, a bad access control implementation uses `tx.origin` to check whether the sender is authorized to perform sensible function calls such as transferring funds. Remember that a contract can interact with other contracts through external calls. An attacker can perform a man-in-the-middle attack by first convincing a victim to make a transaction to a contract controlled by the attacker. After that, the controlled code makes an external call to redirect the transaction to the target, which contains vulnerable `tx.origin` check. This step aims to impersonate the victim. If the vulnerable check is bypassed, then the attacker is able to steal the funds of the victim stored in the target contract. Developers are recommended to use `msg.sender` instead of `tx.origin` for user authentication purposes.

Another example of access control is that the function `selfdestruct` is manipulated by an attacker, which is due to the absence of a proper access control check. Parity designed a multi-signature wallet that interacts with its libraries through an external function call. The library can be destroyed by the owner. Unfortunately, it has a critical security flaw i.e., any user could call the function `initWallet` to gain ownership. In 2017, an attacker namely *devops199* exploited this vulnerability by calling `initWallet` to own the library, then invoked function `selfdestruct` to destroy it. This rendered the library inaccessible from the main wallet, and thus about $300M was frozen and lost forever.

Arithmetic: An arithmetic vulnerability, including integer overflow and integer underflow, occurs when the result of an arithmetic operation goes beyond the range of values that can be represented. For example, considering the data type `uint256`, an overflow occurs when adding one to the maximum value (i.e., $2^{256} - 1$), causing the returned value wraps to become the minimum value 0. In contrast, an underflow occurs when subtracting one from the minimum value (i.e., 0), causing the returned value wraps to become the maximum value $2^{256} - 1$. Arithmetic is an inherent vulnerability, but its consequences are exaggerated in smart contracts. For instance, in April 2018, an attacker exploited an integer overflow vulnerability in a smart contract named `SmartMesh` and stole a massive amount of tokens (i.e., digital currency). Numerous vulnerable smart contracts were identified this way. VeriSmart [66] provided a list of 60 CVEs associated with arithmetic vulnerability.

Unhandled Exception: This vulnerability exists because Solidity lacks uniformity in handling run-time exceptions. When an external function call encounters an exception and fails, the mechanism for handling this exception is determined based on the method used to make the external call. While `transfer` propagates exception back to the caller, other methods such as `send`, `call`, and `delegate` return either `true` or `false`. Because of this inconsistent behaviour, developers often make an assumption that exceptions are properly handled and decide to disregard the returned value of external function calls. An attacker can create a contract that intentionally throws an exception to exploit flawed code or perform a denial-of-service attack. For example, in 2016, a ponzi game King of the Ether Throne (KotET) was deployed. Players send the contract an amount of ether to take "the throne". Upon taking the throne, a reward in ether is sent to the former king. After 24 hours, the current king will receive all the ether in the KoET contract. However, an attacker exploited the unhandled exception vulnerability to prevent other users from becoming a king. That is, the attacker creates a contract that throws an exception in its fallback function. Consequently, every payment transaction sent to the contract will fail and the attacker is the last king of the game to receive all the rewards.

Denial-of-Service: Denial-of-Service (DoS) is a vulnerability that prevents the contract from providing normal services for a while or forever. There are two common ways to launch a DoS attack against smart contracts: *unexpected revert* and *block gas limit*. To perform an *unexpected revert* DoS, an attacker manipulates the external function call of the target contract to ensure that it always throws an exception regardless of users. As a result, the function becomes useless. Another type of DoS attacker is *block gas limit* DoS. A block has an upper gas limit. A transaction is blocked if the cost of the current execution reaches the limit. Therefore, a smart contract itself may suffer from block gas limit even without an attacker. More seriously, if an attacker deliberately manipulates the cost of gas so that the gas limit is always reached, the transaction is terminated in failure.

Frontrunning: This vulnerability is also known as *Transaction order dependency* (TOD). The vulnerability refers to the situation that the outcome of transactions varies depending on the orders in which the transactions are executed. Attackers attempt to manipulate transaction orders to gain advantages. This vulnerability often occurs in decentralized exchanges. That is, an attacker listens to transactions containing large buy orders and attempts to front-running current transactions to raise the token price. For example, in 2022, a user attempted to swap $1.85M worth of Compound cUSDC for USDC on Uniswap but only received $500 [67]. This is because a bot `0xbad` [1] front-run the trade and made $1.02M.

Time Manipulation: In Solidity, developers make use of `block.timestamp` or its alias `now` to get the current timestamp. Smart contracts commonly rely on the timestamp to construct constraints such as locking token sales and unlocking funds. However, miners can deliberately manipulate the timestamp to their advantage. As such, it is advisable for developers to not rely on timestamp to construct critical constraints.

Other Vulnerabilities: Besides the mentioned vulnerabilities, the rapid development of decentralized finance (DeFi) introduces many new vulnerabilities [20] such as *price oracle manipulation*, *sandwich attack*, and *maximum extract value*. The DeFi-related vulnerability detection/repair remains an unsolved problem.

The wide variety of current vulnerabilities and potentially future vulnerabilities have a negative impact on the development of Ethereum, as well as, its ecosystem. We need systematic ways of improving the correctness of smart contracts.

[1] 0xbadc0defafcf6d4239bdf0b66da4d7bd36fcf05a

Part I

Smart Contract Vulnerability Detection

Chapter 3

sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts

Smart contracts are Turing-complete programs that execute on the infrastructure of the blockchain, which often manage valuable digital assets. Solidity is one of the most popular programming languages for writing smart contracts on the Ethereum platform. Like traditional programs, smart contracts may contain vulnerabilities. Unlike traditional programs, smart contracts cannot be easily patched once they are deployed. It is thus important that smart contracts are tested thoroughly before deployment. In this work, we present an adaptive fuzzer for smart contracts on the Ethereum platform called sFuzz. Compared to existing Solidity fuzzers, sFuzz combines the strategy in the AFL fuzzer and an efficient lightweight multi-objective adaptive strategy targeting those hard-to-cover branches. sFuzz has been applied to more than 4 thousand smart contracts and the experimental results show that (1) sFuzz is efficient, e.g., two orders of magnitude faster than state-of-the-art tools; (2) sFuzz is effective in achieving high code coverage and discovering vulnerabilities; and (3) the different fuzzing strategies in sFuzz complement each other.

3.1 Introduction

Nowadays, smart contracts [60], [68] are implemented as Turing-complete programs that execute on the infrastructure of the block-chain [69]. It provides a framework that potentially allows any program (equivalently, contract) to be executed in an autonomous, distributed, and trusted way. Smart contracts thus have the potential to revolutionize many industries. Popular applications of smart contracts include crowd fundraising, online gambling and so on. Ethereum [70], [71] is the first to introduce the functionality of smart contracts. Based on the Ethereum platform, Solidity is the most popular programming language for smart contracts [62].

Like traditional C or Java programs, smart contracts may contain vulnerabilities. Unlike traditional programs, smart contracts cannot be modified easily once they are deployed on the blockchain [72]. As a result, a vulnerability renders the smart contract forever vulnerable, which significantly magnifies the problem. In recent years, there has been an increasing number of news reports on attacks which exploit security vulnerabilities in Ethereum smart contracts. One particularly noticeable example is the DAO attack [73], i.e., an attacker stole more than 3.5 million Ether (which is equivalent to about $45 million USD at the time) exploiting a vulnerability in the DAO contract. To fix the vulnerability, a hard fork was launched which was not only expensive but also caused much controversy [73].

It is thus desirable to develop tools for validating smart contracts to identify vulnerabilities, ideally before they are deployed. Among the range of complementary techniques for validating smart contracts, we focus on automatic testing of smart contracts in this work as testing is often the least expensive and thus the most applicable. To automatically test smart contracts, we must solve the following three problems:

- the test automation problem (i.e., how to run test cases),
- the test generation problem (i.e., what to test),
- and the oracle problem (i.e., what are vulnerabilities).

In the literature, several approaches have been developed for automatic testing smart contracts, each of which answers these three problems in slightly different ways. For instance, ContractFuzzer [26] builds a network with pre-deployed contracts and generates transactions to run smart contracts, generates test cases based on a set of predefined parameter values and targets a set of oracles specific for smart contracts. Oyente [23] runs smart contracts symbolically through symbolic execution, generates test cases for covering different program paths in single functions through constraint solving, and supports multiple oracles to identify 4 kinds of vulnerabilities. teEther [27] similarly applies symbolic execution to generate test cases covering program paths, and focuses on oracles which are related to financial transactions.

In this work, we propose a fully automatic testing engine for smart contracts running on Ethereum called sFuzz. sFuzz is inspired by AFL [37], a well-known fuzzer for C programs, i.e., sFuzz is a feedback-guided fuzzing engine and is inexpensive to apply. sFuzz complements existing testing engines based on symbolic execution like Oyente and teEther, as it is known that fuzzing and symbolic execution are complementary [74], [75]. While AFL-based fuzzing is often effective, it has its limitation as well, i.e., it is often expensive in covering branches guarded with strict conditions. To tackle the problem, sFuzz integrates AFL-based fuzzing with an efficient lightweight adaptive strategy for selecting seeds. Although inspired by search-based software testing [38],

[39], the latter distinguishes itself by having a lightweight objective function (designed considering characteristics of Solidity programs) as well as a novel multi-objective optimization strategy.

sFuzz is built based on Aleth [76] (i.e., an Ethereum VM written in C++), has a system architecture similar to AFL, and is extensible to different Ethereum VMs and oracles as well as fuzzing strategies. sFuzz has been systematically applied to a set of more than 4 thousand smart contracts. The experimental results show that sFuzz is on average more than two orders of magnitudes faster than ContractFuzzer, covers more branches and reveals many more vulnerabilities. A comparison between sFuzz and Oyente shows that they are complementary. Furthermore, experiments with prolonged fuzzing time show that the adaptive strategy improves code coverage. sFuzz is available online and has been adopted by multiple companies.

The remainder of the chapter is organized as follows. Section 3.2 illustrates how sFuzz works through examples. Section 3.3 presents the details of the approach. Section 3.4 shows implementation details of sFuzz. Section 3.5 reports evaluation results. Section 3.6 reviews related work and concludes.

3.2 Illustrative Examples

In this section, we show how sFuzz works step-by-step through two illustrative examples. Note that Solidity source codes for both examples are shown for simplicity. sFuzz requires only the EVM (i.e., Ethereum Virtual Machine) bytecode [70], [71] to fuzz smart contracts.

Given a smart contract, sFuzz automatically configures a block-chain network, deploys the smart contract, and generates multiple transactions each of which calls a function in the contract. The transactions are then executed with

```
pragma solidity ^0.4.20;
contract opposite_game {
  string public question;
  address questionSender;
  bytes32 responseHash;

  function Try(
    string _response ) external payable {
    if(responseHash == keccak256(_response) &&
        msg.value == 100 finney) {
      msg.sender.send(this.balance); } }

  function start_quiz_game(
    string _question, string _answer) public payable {
    if(responseHash==0x0) {
      responseHash = keccak256(_answer);
      question = _question;
      questionSender = msg.sender; } }

  function() public payable {} }
```

FIGURE 3.1: An example with single objective function

an EVM enriched with a set of oracles for identifying vulnerabilities. sFuzz monitors the execution of the transactions to collect certain feedback, e.g., whether a certain branch has been covered and how far the branch is covered. Whenever a vulnerability is revealed, the transactions and the network configuration (i.e., a test case) are saved and reported to the user later on. Otherwise, some of the test cases are selected as *seeds* based on feedback collected during the transaction execution according to certain seed selection criteria. Afterwards, the *seeds* are mutated to generate the next generation of test cases. This process repeats until a time out occurs.

In the following, we describe how sFuzz works using the contract shown in Figure 3.1. The contract implements a simple quiz game. The contract is based on contract *opposite_game*[1] with minor modification for simplicity. A quiz can be created by calling function *start_quiz_game*. The response is hashed and

[1]address: 0x467532e79222670a2044c9b168bcbaa33b390ef5

then saved in the $responseHash$ variable. The user then calls the try function with their answer as the argument and pays a fee of $100\ finney$ (which is a unit of the token) for each try. If the answer is correct, a reward is sent to the user.

This contract suffers from a vulnerability known as *Gasless Send* when line 11 is executed and a costly *fallback function* is called. That is, when function $send()$ at line 11 is executed, if the receiver is a contract, its fallback function is executed automatically. Because function $send()$ only forwards 2300 units of gas (i.e., price to pay for executing the function), an *out-of-gas* exception is thrown if the fallback function is costly (e.g., costs more than 2300 units of gas). In this case, the $send()$ function simply returns $false$ and because the returned value is not checked and handled accordingly, the owners of the contract can keep the reward for themselves.

To expose this vulnerability, first a network is configured with several addresses and associated balances. This contract is then deployed at one of the addresses. In addition, an attacker contract with a costly fallback function is deployed automatically. To expose the vulnerability, a test case (i.e., a sequence of transactions) with such a network configuration must first call function $start\_quiz\_game$ and then function Try with parameters such that all 2 conditions in function Try at line 9 and 10 are satisfied. The condition at line 9 is satisfied with a test case that sets all the parameters and contract variables to the default value of 0. Note that $responseHash$ is set to $keccak256(\_answer)$ at line 16 and is compared to $keccak256(\_response)$ at line 9. However, generating a test case which satisfies the second condition by randomly generated test values is highly unlikely. The variable $msg.value$ has a size of 32 bytes and thus we have only $\frac{1}{2^{256}}$ probability to generate the value 100 (if we generate random values with a uniform distribution among all possible values). Existing fuzzing strategy in AFL is ineffective in this case as well, i.e., AFL selects test cases that cover new branches as $seeds$. Since all test cases generated through mutation are unlikely to cover the then-branch at line 10, they are equally 'bad' according to the AFL seed selection strategy.

sFuzz complements AFL's seed selection strategy with an adaptive strategy that prioritizes the seeds according to a quantitative measure (i.e., a distance) on how far a seed is from covering any just-missed branch. For this example, the distance for covering the just-missed branch (i.e., the then-branch) is computed as: $|msg.value - 100| + 1$, based on the value of $msg.value$ when the branch at line 10 is reached in the test case. Intuitively, the smaller the distance is, the closer the test case is to cover the branch (i.e., with a $msg.value$ closer to 100). In particular, when $msg.value$ is exactly 100, the distance value reaches the minimum value of 1. Based on this measurement, sFuzz iteratively selects $seeds$ which gradually gets closer and closer to satisfying the condition at line 10. In our experiment, after 140 generations, sFuzz generates a test case which covers the branch, and reveals the vulnerability.

The above example shows a simplistic situation where there is only one just-missed branch. In general, there may be multiple just-missed branches and thus sFuzz measures a distance for each pair of test case and just-missed branch, i.e., how far is the branch from being covered by the test case. Then for each just-missed branch, sFuzz selects the test case with the minimum distance as the $seed$. For instance, the contract in Figure 3.2 shows a function which performs some basic arithmetic operations. There are two different branches, i.e., the condition at line 5 for comparing y with 110 and the one at line 6 for comparing y with 10010. Assume that both then-branches are yet to be covered. Given any test case, sFuzz computes two distances, one for covering the first then-branch;

```
1 pragma solidity ^0.4.20;
2 contract multiple_objective_function {
3   function foo(int x) {
4     int y = x*x + 10;
5     if(y == 110) { ... }
6     if(y == 10010) { ... } } }
```

FIGURE 3.2: An example with multiple objective functions

and the other for covering the second then-branch. Given a set of test cases, sFuzz selects, for each of these two branches, a test case which has minimum distance as seed, to generate further test cases. After repeating the process multiple times, sFuzz generates two test cases that cover the two then-branches. *We remark that for this example, due to the non-linear computation at line 4, approaches based on symbolic execution like Oyente [23] and teEther [27] are ineffective due to the limitation of underlying constraint solvers.*

3.3 Fuzzing Smart Contracts

In this section, we define our problem and then present our approach in detail step-by-step.

3.3.1 Problem Definition

A smart contract $\mathcal{S}$ typically has a number of instance variables, a constructor and multiple functions, some of which are public. It can be equivalently viewed in the form of a control flow graph (CFG) $\mathcal{S} = (N, i, E)$ where N is a finite set of control locations in the program; $i \in N$ is the initial control location, i.e., the start of the contract; and $E \subseteq N \times C \times N$ is a set of labeled edges, each of which is of the form (n, c, n') where c is either a condition (for conditional branches like if-then-else or while-loops) or a command (i.e., an assignment). Note that for simplicity, we define the smart contract as one single graph rather than defining one graph for each function and then connecting them through a call graph. A node in the graph is branching if and only if it has multiple child nodes and its outgoing edges are labeled with conditions. We refer to an outgoing edge of a branching node as a branch.

Test cases: A test case for $\mathcal{S}$ is a pair (σ_0, Σ) where σ_0 is a configuration of the blockchain network and Σ is a sequence of transactions (i.e., function calls). The configuration σ_0 contains all information on the setup of the network which is relevant to the execution of the smart contract. Formally, σ_0 is a tuple (b, ts, SA, SB, v) where b is the current block number, ts is the current block timestamp, SA is a set of the addresses of the smart contracts (including the smart contract under test as well as other invoked contracts), SB is a function which assigns an initial balance to each address and v is the initial valuation of the persistent state. $\Sigma = \langle m_0(\overrightarrow{p_0}), m_1(\overrightarrow{p_1}), \cdots \rangle$ is a sequence of public function calls of the smart contract under test, each of which has an optional sequence of concrete input parameters $\overrightarrow{p_i}$. Note that m_0 must be a call of the constructor.

The task of fuzzing a smart contract is thus to generate a set of test cases (a.k.a. test suite) according to certain testing criteria. The execution of a test case t traverses through a path in the CFG $\mathcal{S}$, which visits a set of nodes and edges. For simplicity, we assume that one test execution covers one unique path (i.e., there is no non-determinism). Furthermore, a trace generated by t is a sequence of pairs of the form $\langle(\sigma_0, n_0), (\sigma_1, n_1), \cdots\rangle$ where $(n_0, n_1, \cdots)$ is the sequence of nodes visited by t and σ_i is the configuration at the time of visiting node n_i for all i.

Code Coverage: Ideally, we aim to generate a test suite which reveals all vulnerabilities in the contract. However, as we do not know where the vulnerabilities are, we must instead aim to achieve something more measurable. In this work, our answer is to focus on code coverage, in particular, branch coverage. We remark that our approach can be extended to support different coverage at the cost of additional code instrumentation. A branch in $\mathcal{S}$ is covered by a test suite if and only if there is a test case t in the suite that visits the edge at least once. The branch coverage of a test suite is calculated as the percentage of the covered branches over the total number of branches. Note that identifying the total number of (feasible) branches statically in a smart contract is often infeasible for two reasons. First, some branches might be infeasible (i.e., there does not exist any test case that visits the branch) and knowing whether a branch is feasible or not is a hard problem. Second, EVM has a stack-based implementation which makes identifying all potentially feasible branches hard (as we will explain in more detail in Section 3.4). *Our problem is thus reduced to generate a test suite which maximizes the number of covered branches.*

To achieve maximum code coverage, one way is to generate a large test suite (e.g., through random test generation). However, in practice, we often have limited resources (in terms of time or the number of computer processes) and thus our problem is refined as '*to generate a test suite which maximizes the number of covered branches as efficiently as possible*'. Our solution to the problem is feedback-guided adaptive fuzzing.

Fuzzing is one of the most popular methods to create test cases [77]. A feedback-guided fuzzing system (a.k.a. fuzzer) takes a program under test and an initial test suite as input, monitors the execution of the test cases to obtain certain feedback, generates new test cases based on the existing ones in certain ways and then repeats the process until a stopping criteria is satisfied. We present details of our feedback-guided adaptive fuzzing process in Section 3.3.2.

Oracles: The remaining problem is then how to tell whether a test case reveals a vulnerability. In this work, we adopt a set of oracles from previous approaches [23], [26] including *Gasless Send*, *Exception Disorder*, *Timestamp Dependency*, *Block Number Dependency*, *Dangerous DelegateCall*, *Reentrancy*, *Integer Overflow/Underflow*, and *Freezing Ether*. We refer the readers to Section 3.4 for details.

3.3.2 Feedback-Guided Adaptive Fuzzing

The general idea of feedback-guided fuzzing is to transform the test generation problem into an optimization problem and use some form of feedback as an *objective function* in solving the optimization problem. Our fuzzing strategy is adaptive as we change the objective function adaptively based on the feedback. At the top level, sFuzz employs a genetic algorithm [78] which is inspired by the well-known AFL fuzzer to evolve the test suite in order to iteratively improve its branch coverage.

The overall workflow is shown in Algorithm 1. Variable $suite$ is the test suite to be generated. It is initially empty. Whenever a test case covers a new branch, it is added into $suite$. Variable $seeds$ is a set of seed test cases, based on which new test cases are generated. First, we generate an initial test suite using function $initPopulation()$. The loop from line 3 to 6 then iteratively evolves the test suite. In particular, we add those test cases in $seeds$ which cover new branches

Algorithm 1: The test generation algorithm

```
1 let $suite$ be an empty test suite;
2 let $seeds := initPopulation()$;
3 while not time out do
4     add tests in $seeds$ which covers new branches into $suite$;
5     let $seeds := fitToSurvive(seeds)$;
6     let $seeds = crossoverMutation(seeds)$;
7 return $suites$;
```

(i.e., any branch which is not covered by test cases in $suite$) into $suite$ at line 4. At line 5, we filter the test cases in $seeds$ through function $fitToSurvive()$ so as to focus on those seeds which are more likely to lead to test cases covering new branches later. At line 6, function $crossoverMuatation()$ generates more test cases based on the test cases in $seeds$. The loop continues until a pre-set time out is triggered. While Algorithm 1 resembles the one in AFL, the differences are in the details of each function. In the following, we present each function in detail.

Generating Initial Population: Function $initPopulation()$ generates an initial population containing multiple test cases. As mentioned above, to generate a test case, we need to generate an initial configuration σ_0 as well as a sequence of (public) function calls with concrete parameters. The initial configuration by default is as follows (in hexadecimal): $b = 0$, $ts = 0$, $SA = \{\texttt{0xf0}\}$, $SB = \{\texttt{0xff00...}\}$ and v is set using the declared initial value for each variable representing the persistent state. sFuzz additionally allows a user to customize the initial configuration, i.e., the user is allowed to provide an initial set of test cases.

Next, we generate multiple sequences of transactions, each of which is a function call with concrete parameters. For a contract with n functions, we

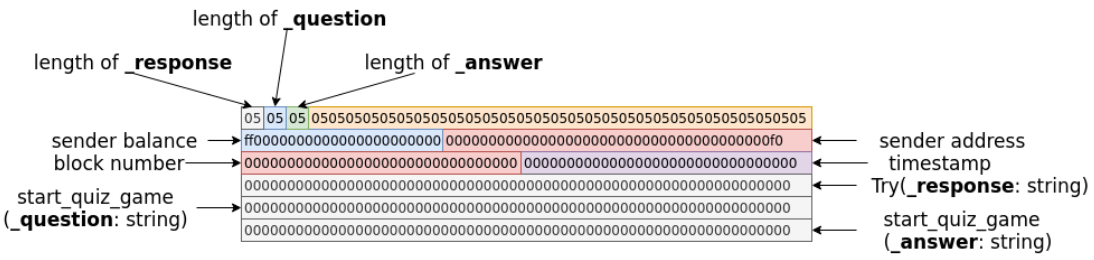


FIGURE 3.3: A generated test case

generate n sequences. In each sequence, a different function is called once after the constructor is called. This makes sure that each function is tested at least once (i.e., function coverage is 100%).

For each function call, we generate a random value for each parameter based on its type. Note that if the parameter type has a fixed-length, e.g., of type $uint256$, this is straightforward. If the type does not have a fixed length (e.g., an array or a string), we first randomly generate a number (with a range from 0 to $bound$ where $bound$ is a bound on maximum length with a default value of 255) representing the number of elements in the parameter (e.g., number of characters) and then generate a corresponding number of element values.

Each test case is encoded in form of a bit vector. In the terminology of genetic algorithms, such bit vectors can be naturally regarded as *chromosomes*. The size of the bit vector equals to the number of bits for encoding the configuration plus the number of bits encoding the function calls. Note that for each test case, we keep a list of function calls (which always includes the constructor in the contract) and then encode each parameter value. If the parameter value is of variable-length, we use $\lceil \log bound \rceil$ (where $bound$ is a bound on the maximum length with a default value of 255) to encode the length of the parameter value. For example, given the contract shown in Figure 3.1, (part of) the encoding of a test case is shown in Figure 3.3 where each part of encoding is labeled in the figure. It contains 192 bytes, of which the first 96 bytes are initial configuration and the last 96 bytes are a sequence of two function calls and the corresponding input parameters. As there are three string parameters, the first 3 bytes including `0x05`, `0x05` and `0x05` encode the length of `_response`, `_question` and `_answer` respectively. The remaining `0x05` values are used when there are more than 3 dynamic variables.

Before executing the test case, the bit vector is decoded to a test case according to our internally defined protocol. Note that the bits in the bit vector may be correlated with each other in multiple ways. For instance, the bits presenting the length of a variable-length value must be equal to the 'length' of the value.

Fitness After executing the $seeds$ at line 4 in Algorithm 1, function $fitToSurvive()$ is called to evaluate the fitness of the $seeds$ according to a fitness function. Note that the fitness function plays an extremely important role.

In sFuzz, we combine two complementary strategies. One is adopted from AFL, which works as follows. While $seeds$ are executed, sFuzz monitors the execution and records the branches that each test case cover. A test case is deemed 'fit to survive' if it covers a new branch in the contract, e.g., a branch which is not covered by any test case in $suite$. This strategy has been shown to be effective in many settings [37] and indeed our experimental results show that it is effective in covering most of the branches (see Section 3.5).

Although the AFL strategy allows us to quickly cover most of the branches, it often makes very slow progress in covering the remaining ones afterwards, i.e., often those branches which are with strict conditions. The reason is that most likely the randomly generated test cases would fail to satisfy the strict condition. In such a case, the above fitness function offers little feedback and guideline on how to generate new test cases. For instance, the probability of satisfying the second condition at line 10 of Figure 3.1 is as low as $\frac{1}{2^{256}}$ (if we assume that every value is equally likely to be generated). Intuitively, however, it is clear that a test input with $msg.value = 200$ is 'closer' to satisfy the condition than a test input with $msg.value = 10000000$. sFuzz thus integrates an adaptive strategy which selects $seeds$ based on a quantitative measure on how far a test case is from covering any just-missed branch.

Let br_n be a just-missed branch in $\mathcal{S}$, i.e., an uncovered outgoing edge from a branching node n in $\mathcal{S}$ and n has been covered. The idea is to define a function $distance(t, br_n)$ where t is a test case to return a quantitative measure on how far the branch br_n is from being covered by t.

Assume that br_n is labeled with a condition c. Note that c can be either $true$, $false$, $a == b$, a != b, $a >= b$, $a > b$, $a <= b$, or $a < b$ at the byte-code level where a and b are variables or constants. In our setting, since br_n is assumed to be a just-missed branch, c must not be $true$ (otherwise br_n must be covered already). Function $distance(t, br_n)$ is then defined as follows.

$$distance(t, br_n) = \begin{cases} K & \text{if } c \text{ is } false \\ \mid a - b \mid + K & \text{if } c \text{ is } a == b \\ K & \text{if } c \text{ is } a \text{ != } b \\ b - a + K & \text{if } c \text{ is } a >= b \text{ or } a > b \\ a - b + K & \text{if } c \text{ is } a <= b \text{ or } a < b \end{cases}$$

where K is a constant which represents the minimum distance. It is set to be 1 in sFuzz. Intuitively, $distance(t, br_n)$ is defined such that the closer the branch is from being covered, the smaller the resultant value is.

With the above, function $fitToSurvive(seeds)$ then selects the seeds as shown in Algorithm 2. The loop from line 2 to 4 goes through every test case to select

Algorithm 2: Algorithm $fitToSurvive(seeds)$

```
1  let $newSeeds$ be an empty set of test cases;
2  foreach $seed$ in $seeds$ do
3      if $seed$ covers a new branch then
4          add $seed$ into $newSeeds$;
5  foreach uncovered branches $br_n$ do
6      let $min$ be $+\infty$; let $t$ be a dummy test case;
7      foreach $seed$ in $seeds$ do
8          if $distance(t, br_n) < min$ then
9              let $min$ be $dist(t, br_n)$;
10             let $t$ be $seed$;
11     add $t$ into $newSeeds$;
12 return $newSeeds$;
```

those which cover a new branch. Afterwards, for each just-missed branch br_n in the smart contract, the loop from line 5 to line 11 selects a test case from $seeds$ which is the closest to cover the branch according to $distance(t, br_n)$. Note that one seed is selected for each just-missed branch, which makes this algorithm a lightweight multi-objective optimization approach. All selected seeds are then used for crossover and mutation to generate more test cases in the next step. We refer the readers to Section 3.2 for an example.

Remark: The above-described strategy is inspired by search-based software testing (SBST) [38], [39] and yet it differs from SBST in several ways. The high-level reason for the difference is that having an AFL-based approach for fuzzing requires us to run test cases efficiently whereas existing SBST's seed selection strategy is time-consuming. Furthermore, due to the stack-based implementation of EVM, implementing existing the SBST strategy is infeasible. In the following, we present the differences in detail.

First, existing state-of-the-art SBST techniques (i.e., the one in EvoSuite [38]) measures how far a test case t is from covering any uncovered branch (not only those just-missed ones) in a more complicated way. That is, given CFG $\mathcal{S} = (N, i, E)$, let the distance from a node n_1 to node n_2 to be the minimum number of edges along any path from n_1 to n_2. Let br_n be any uncovered branch and m be a node covered by t which is the nearest node to n, i.e., m has a minimum distance to n compared to any other node covered by t. SBST uses the following function to measure how far t is from covering br_n.

$$dist(t, br_n) = appr\_dist(t, br_n) + norm(distance(t, br_m))$$

where br_m is an outgoing edge of m which is along the shortest path from m to n. Note that if m is n (i.e., in case br_n is just-missed), br_m is simply br_n. Function $appr\_dist(t, br_n)$ is a measurement of how far branch br_n is from being covered by test case t, i.e., the distance from m to n plus 1. For instance, given a control flow graph as in Figure 3.4, if t covers only the edge $A \rightarrow B \rightarrow E$, $appr\_dist(t, C) = 1$ since there is one branch from B to reach C and there are two branches from A to reach C via D. Similarly, $appr\_dist(t, F) = 2$. Lastly, function $norm(x)$ is a normalization function which normalizes the results of $distance(t, br_m)$ to a value between 0 and 1. One such function is $norm(x) = 1 - 1.001^{-|x|}$ [38].

Applying the above strategy in fuzzing Solidity smart contracts is inefficient, if not infeasible, for multiple reasons. First, calculating $appr\_dist(t, br_n)$ would require us to construct the complete CFG. However, constructing the CFG based on bytecode only is highly nontrivial. In EVM, branches are realized with the opcode *jumpi*, with a value representing the target program counter dynamically at runtime. The only way to know the target is to fully simulate the stack, which is expensive. Second, even if we have the CFG, computing $appr\_dist(t, br_n)$ is still expensive. Given a CFG with K uncovered nodes. To maintain a list of 'best' test cases for each uncovered node, we have to calculate $appr\_dist(t, br_n)$ for all K uncovered nodes, i.e., by building a table of the shortest paths from all nodes to these K nodes. Furthermore, whenever a new node is covered, $appr\_dist(t, br_n)$ must be updated. The overhead is unreasonable given that efficiency is key for AFL-based fuzzing. By focusing on just-missed branches, sFuzz avoids both problems. That is, $appr\_dist(t, br_n)$ is always 1 for any just-missed branch br_n since node n must have been covered. Furthermore, because it is constant for any uncovered branch, we can simply skip it in $dist(t, br_n)$ and so that $dist(t, br_n)$ is reduced to $distance(t, br_n)$, without even the need to normalize. This further reduces the overhead.

Another key difference between sFuzz's strategy and existing SBST's is the multi-objective searching strategy. The multi-objective search strategies in existing SBST consider each uncovered branch as an objective and select Pareto-optimal seeds to evolve in next generation. Given a set of uncovered branch $\{b_1, b_2, ..., b_m\}$, a set of seeds $\{t_1, t_2, ..., t_n\}$, we say t_i is more Pareto-optimal than t_j if $\forall k \in 0..m$, $distance(t_i, b_k) < distance(t_j, b_k)$. Otherwise, we say that t_i and t_j are Pareto-equivalent. All Pareto-equivalent seeds form a Pareto frontier and the seeds can fall into several Pareto frontiers. Existing SBST selects the most Pareto-optimal seeds to evolve. A known problem for such a strategy [79] is that the number of seeds in the same Pareto frontier soars with the increase of the number of objectives (i.e., uncovered branches). For example, there could be hundreds of seeds in the most Pareto-optimal frontier with only 3-5 objectives, which makes it hard to select the most promising seeds and increases the runtime overhead. In contrast, sFuzz keeps one best seed for each just-missed branch (line 6–11 in Algorithm 2) and as a result, the number of seeds remains

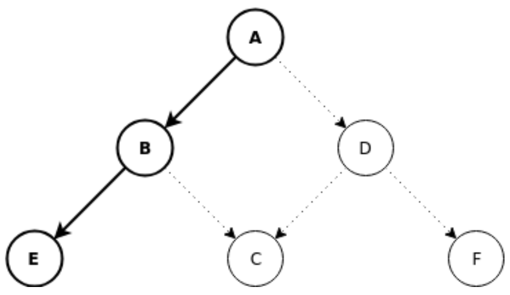


FIGURE 3.4: A control flow graph

small (i.e., equivalent to the number of just-missed branches). Our experimental results show that such a strategy balances effectiveness in identifying good seeds and efficiency well.

3.3.3 Crossover and Mutation

Function $crossoverMutation()$ generates new test cases based on those in $seeds$ through crossover and mutation. sFuzz adopts all of the crossover strategies from AFL and introduces news ones specific for smart contracts. Furthermore, due to correlation between parameters of a test case, sFuzz additionally makes sure the generated test cases are valid. For instance, sFuzz (1) randomly chooses two test cases from $seeds$; (2) breaks the two test cases into two pieces at a selected position; and (3) swaps the second pieces to form two new test cases. Note that due to correlations between the bits representing a test case, there is no guarantee that the resultant test cases are valid and thus sFuzz always checks for validity and discard those invalid ones.

Mutation is another way of generating new test cases. Given a *seed* encoded in the form of a bit vector, sFuzz supports a set of mutation operators to generate new test cases. All mutation operators are shown in Table 3.1.

Recall that a test case is in the form of an initial configuration and a sequence of function calls with concrete parameters. The first three mutation operators

TABLE 3.1: Mutations for fix-length values

Name
pruneMethodCall (new)
addMethodCall (new)
swapMethodCall (new)
singleWalkingBit, twoWalkingBit, fourWalkingBit, 1/2/4 consecutive bits
singleWalkingByte, twoWalkingByte, fourWalkingByte
singleArith, twoArith, fourArith
singleInterest, twoInterest, fourInterest
overwriteWithDictionary
overwriteWithAddressDictionary

aim to alter the sequence of function calls, by pruning a function call, adding a function call or swapping two function calls. When a function call is pruned (or added or swapped), the corresponding concrete parameters are pruned (or added or swapped) accordingly.

For those values in a test case other than those representing the called functions, sFuzz categorizes them into two groups. The first group contains those values which have fixed-length (e.g., a parameter of type $uint256$). sFuzz systematically applies the remaining mutation operators shown in Table 3.1 to generate new values, which are inspired by the mutation operators in AFL. Note that account addresses (and balances) are handled slightly differently (refer to the last row in the table) as there are special format requirements. Each address has 32 bytes, in which the last 20 bytes contain the address value and the first 12 bytes contain the balance of the address. For instance, the value `0xff00...00...00f0` represent an address `0xf0` with balance `0xff000000000000000000000`.

The second group contains those values which have variable-length (e.g., a parameter of type $array$). For such values, their lengths are encoded as part of the test case as well. We thus first mutate the value representing the length in such a way that the result is a random value between 0 and 255 where 255 is an upper bound. If the new length is less than the current one, the corresponding value is shortened accordingly by pruning the additional bits. If the length is more than the current one, random type-compatible values are padded accordingly.

Note that we discard identical test cases generated through either crossover or mutation. Furthermore, although we do not set a limit on the number of mutations generated from a test case, we apply multiple heuristics adopted from AFL to reduce the number of mutations. For instance, if applying the *WalkingByte* mutation to a block of 32 bytes does not result in any test case which covers a new branch, in the next stages sFuzz will not mutate that block. We refer the readers to AFL for details on these heuristics [37].

3.4 Implementation

sFuzz is implemented in C++ with an estimated 4347 lines of code. It is publically available (https://sfuzz.github.io). It has 3 main components: *runner*, *libfuzzer* and *liboracles*.

Runner: manages the execution of the test cases. sFuzz takes as input the bytecode of a smart contract along with the ABI (i.e., application binary interface, which can be generated automatically using existing tools) of the contract. The *runner* then generates a bash script file which contains a list of commands to analyze the ABI, and set options for the other two components.

The *runner* sets up a test network based on which smart contracts are deployed and transactions are executed. To generate test cases for functions with address-type parameters, sFuzz deploys a pool of externally owned accounts in the test network with random balances. The pool size is less than or equal to the number of address-type parameters because it is possible to set the same address to multiple address-type parameters. The values for address-type parameters are then chosen randomly from this pool. In addition, sFuzz deploys two special smart contracts as attackers, i.e., a *normal attacker* and a *reentrancy attacker*. Each attacker is set as the owner of the contract under test in turn. The *normal attacker* throws an exception whenever other contracts call its payable fallback function. The *reentrancy attacker* calls back the function which makes a call to its payable fallback function. If the attacker fails to call back, it acts as a *normal attacker*. Note that the *reentrancy attacker* is only loaded to detect *Reentrancy* vulnerability. Otherwise, the *normal attacker* is loaded to avoid call loops of *Reentrancy Attacker* which significantly reduces the speed of sFuzz.

Libfuzzer: solves the test generation problem, i.e., how to selectively generate test cases, by implementing the fuzzing strategy presented in the previous sections. It is responsible for multiple tasks.

First, it constructs the CFG of the given smart contract on-the-fly. Ideally, we would like to construct the CFG statically before fuzzing. However, constructing the CFG based on bytecode only is highly nontrivial. In EVM, branches are realized with the opcode *jumpi*, with a value representing the target program counter dynamically at runtime. The only way to know the target is to fully simulate the stack, which is expensive. Therefore, sFuzz constructs the CFG on-the-fly while fuzzing. That is, whenever the opcode *jumpi* is executed, the two destinations are recorded. If these two destinations are not part of the CFG yet, two new nodes are created accordingly representing the two destinations in the CFG.

Second, component *libfuzzer* implements the fuzzing algorithm discussed in Section 3.3. One optimization is that we identify *view* functions (i.e., those which do not change any variables) and exclude them from test case generation. The justification is that these view functions do not change the states and having them does not additionally expose those vulnerabilities sFuzz targets at (see below). Note that view functions are marked by `view`, `pure` or `constant` keywords, sFuzz reads ABI file to recognize them.

Liboracles: solves the oracle problem, i.e., it monitors the execution of a test case and checks whether there is a vulnerability according to an extensible library of oracles used in sFuzz. sFuzz monitors the execution of test cases through the hooking mechanism supported by EVM. Whenever EVM executes an opcode, it creates an event containing read-only execution information, such as the values of the stack, memory, program counter, and the current executed opcode. sFuzz monitors these events for constructing the CFG and computing $distance(t, br_n)$, as well as logs the events for vulnerability detection. To reduce the execution overhead, vulnerability detection is conducted offline in batches (i.e., once for every 500 test cases). This design allows sFuzz to easily support different versions of Solidity, i.e., by simply replacing the EVM packed in sFuzz.

sFuzz has an extensible architecture which allows it to easily support different oracles as well. Currently, sFuzz supports 8 oracles inspired by the previous work [23], [26]. Since these oracles are not our main contribution, we refer the readers to [23], [26] for details.

These oracles are checked based on the logs of test cases. For instance, to check if a test case expose the *Gasless Send* vulnerability, we check that whether test case executes a *CALL* instruction with some data greater than 0 when the gas is equal to 2300. The test cases that expose vulnerabilities in the contract are kept in a separate test suite and reported to the user together with the vulnerabilities that they expose. Note that by design, sFuzz always reports true positives according to our definition of vulnerability except in the case of *Freezing Ether*. However, in practice, a reported vulnerability might be a false positive as it may be what the user intended (i.e., our definition of vulnerability is too strict). In the case of *Freezing Ether*, the identified 'warning' might be a false positive if there exist some test cases which call $send()$ or $transfer()$ but such test cases are never generated. Technically, the problem of checking whether there is *Freezing Ether* vulnerability can only be solved if we cover all feasible opcode (which is often infeasible).

3.5 Experiments and Evaluation

In this section, we evaluate sFuzz through multiple experiments. The experiments are designed to answer the following research questions (RQ).

- *RQ1: How efficient is sFuzz?*
- *RQ2: Is sFuzz effective in finding smart contract vulnerabilities and obtaining high code coverage?*
- *RQ3: Is the adaptive strategy useful?*

Our test subjects include 4112 smart contracts which we collect from EtherScan [80]. These contracts are implemented using Solidity 4.2.24, which is the most popular version of Solidity. Moreover, the source code for these contracts are available, which makes the evaluation more accurate. We note that sFuzz can run with bytecode only. For a baseline comparison, we compare sFuzz with a fuzzer named ContractFuzzer reported in [81] and a symbolic execution tool named Oyente reported in [23]. Other fuzzers for smart contracts have been mentioned in [27]. However, we fail to find the reported tools online or through the authors. We run the experiments 3 times and report the average as the result. All experimental results reported below are obtained on an Ubuntu 18.04.1 LTS machine with Intel Core i7 and 16GB of memory. We use the default initial configuration as presented in Section 3.3.2.

3.5.1 Efficiency

To answer RQ1, we systematically apply sFuzz, ContractFuzzer and Oyente on all 4112 smart contracts. To save time, each contract is run for 2 minute in this experiment. Note that in general the adaptive fuzzing strategy takes time to show its effectiveness (as we will show later) and thus this setting gives an edge to other tools.

We measure the efficiency of sFuzz by counting how many test cases are generated and executed per second. Naturally, a test case for a more complicated contract (e.g., with many loop iterations) takes more time to execute. Thus, we show how efficiency varies for different contracts. Figure 3.5 summarizes the result, where each bar represents 10% (about 400) of the fuzzed contracts and the y-axis shows the number of test cases generated and executed per second. The contracts are sorted according to how efficiently it can be fuzzed. From the figure, we observe that the efficiency varies significantly over different contracts, i.e., sFuzz generates and executes more than 989 test cases per second on average for the top 10% of the contracts, and less than 14 test cases for the bottom 20%. On average, sFuzz generates and executes more than 208 test cases per second.

Figure 3.5 also compares the efficiency of sFuzz with Oyente and Contract-Fuzzer. From the results, we observe that sFuzz is significantly more efficient than other tools. On average, ContractFuzzer and Oyente generate and execute 0.1 and 16 test cases per second respectively. There are multiple reasons why sFuzz is much faster. First, ContractFuzzer simulates the whole network and manages the blockchain (e.g., commit state changes to storage and append new mined blocks to blockchain after function calls), whereas sFuzz simulates only details of network or blockchain which are relevant to vulnerabilities in smart contracts. Second, sFuzz has a highly optimized implementation in C++, whereas ContractFuzzer is based on Node.js and Go language. In the case of Oyente, because it is a symbolic execution tool, Oyente is expected to run slower than a fuzzer like sFuzz.

We further conduct an experiment to measure the overhead of monitoring the execution of a test case (using the hooking mechanism) and the overall overhead of the fuzzing process (including the overall of monitoring the execution, constructing the CFG, mutating the test cases and comparing them, etc.). We apply sFuzz to a set of 60 randomly selected contracts and measure the time spent on executing the test cases, monitoring the execution and other steps of the fuzzing process. The results show that on average the monitoring consumes about 10% of the total execution time and the overhead of the fuzzing process (including monitoring) is about 14%. This is very efficient compared to the reported overhead in other fuzzers [75].

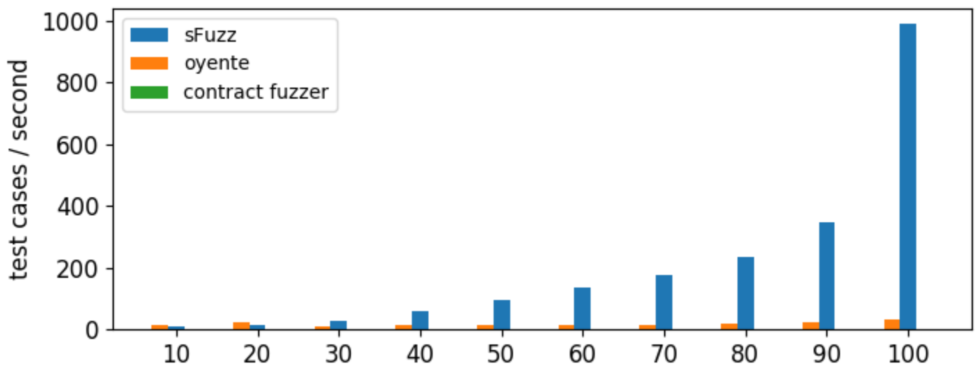


FIGURE 3.5: Efficiency comparison between sFuzz, Oyente, and ContractFuzzer

3.5.2 Effectiveness

To answer RQ2, we aim to measure the branch coverage achieved by the test suite generated for each smart contract, as well as count the number of vulnerabilities identified. However, measuring branch coverage precisely is highly non-trivial due to, for instance, the problem of infeasible branches. Thus, we instead measure the number of distinct branches covered by the generated test suite. Figure 3.6 summarizes a comparison between sFuzz and ContractFuzzer in terms of the number of distinct branches covered. The y-axis is the number of branches covered by sFuzz minus that of ContractFuzzer and each point on the x-axis represents a smart contract. The contracts are sorted by their y-axis value. Similarly, Figure 3.7 shows the comparison between sFuzz and Oyente.

For most of the smart contracts (i.e., 4077 of 4112 contracts) sFuzz covers more branches than ContractFuzzer. To our surprise, ContractFuzzer managed to cover more branches for 35 contracts. A closer investigation shows that the number of branches covered by ContractFuzzer is inflated for the following reasons. First, as sFuzz does not execute *view* functions (for efficiency reasons), all branches in these functions are not counted. Because *view* functions

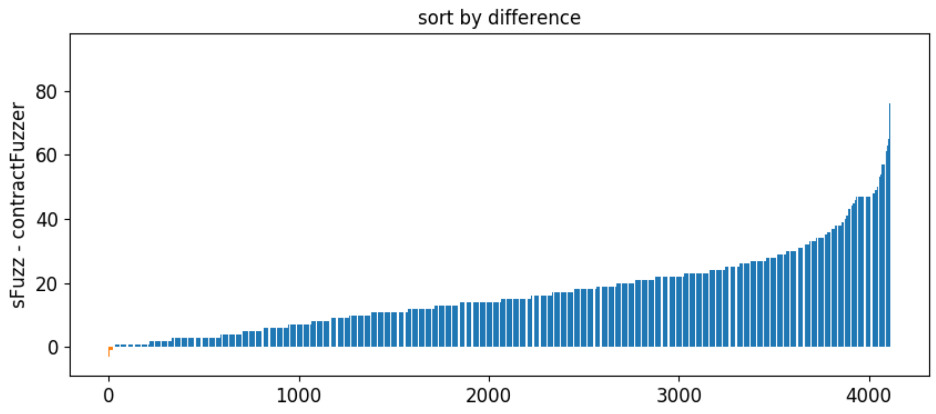


FIGURE 3.6: Coverage comparison between sFuzz and ContractFuzzer

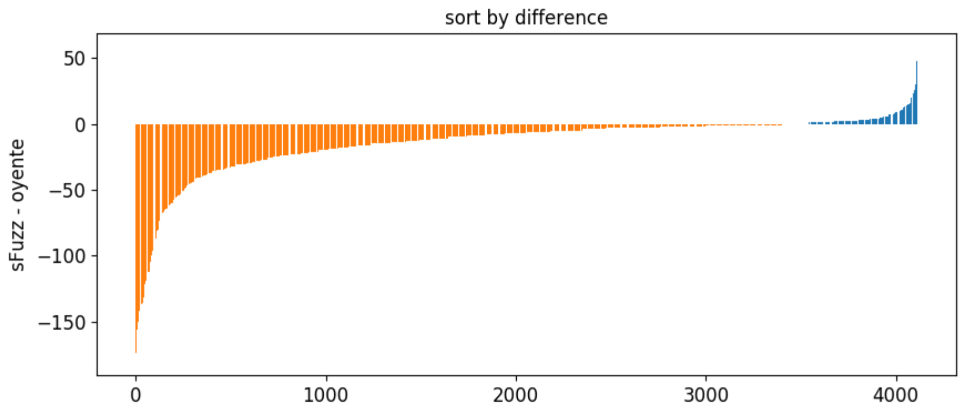


FIGURE 3.7: Coverage comparison between sFuzz and Oyente

do not modify the state of a smart contract, they are considered irrelevant to vulnerabilities. Second, ContractFuzzer sometimes generates invalid test cases which fail mandatory constraints and cover additional branches. Mandatory constraints are generated by the compiler (i.e., the Solidity compiler) and are embedded in the bytecode to assert the correctness logic of function calls or data types. For example, ContractFuzzer invokes a *fallback* function of a non-fallback contract or sends Ethereum to functions which are not marked with the *payable* keyword. As a result, the mandatory constraints are failed which lead to branches which signal an error in the test case being covered.

```
contract A {
  mapping(address => uint) balances;
  uint id = 10;
  function main(uint x, uint y) {
    if (id == 9) {
      if (balances[msg.sender] > 10) {
        uint sum = x + y; } } } }
```

FIGURE 3.8: Oyente visits infeasible branches

In the case of Oyente, in 3402 contracts, Oyente covers more branches than sFuzz. An investigation shows that Oyente analyzes every function separately and thus has to assume that state variables can take arbitrary values (without considering their initial values or constraints on how the values are updated). As a result, Oyente can easily satisfy almost all conditions in smart contracts. Given the sample contract A in Figure 3.8, Oyente covers 99.1% EVM code and discovers an integer overflow vulnerability. It means that these conditions: $id == 9$ and $balances[msg.sender] > 10$ are satisfied. However, it is impossible as there is no way to change values of id and $balances[msg.sender]$. Often, a condition in smart contract is the comparison between local/parameter variables and state variables, e.g., $balances[msg.sender] > value$ (whether sender has enough Ethereum to deduce). In such cases, sFuzz must call the function which sets certain values to the state variables before satisfying them whereas Oyente assigns arbitrary values directly to state variables. It is apparent to us that Oyente's approach is flawed and would 'cover' many infeasible paths.

In the following, we summarize the number of vulnerable contracts discovered by sFuzz in each category. The results are shown in Table 3.2. The first column shows the type of vulnerability. The next three columns show the number of vulnerable contracts found by sFuzz, ContractFuzzer and Oyente respectively. The sub-column # show the number of contracts that have the vulnerability according to each vulnerability type and the second sub-column is the

TABLE 3.2: Vulnerabilities

Vulnerability Type	**sFuzz**		**ContractFuzzer**		**Oyente**	
	#	**true posi.**	**#**	**true posi.**	**#**	**true posi.**
Gasless Send	764	100%	14	100%	0	N.A.
Exception Disorder	36	100%	6	100%	0	N.A.
Reentrancy	29	100%	3	100%	52	60%
Timestamp Dependency	243	86%	28	86%	102	100%
Block Number Dependency	59	80%	16	95%	0	N.A.
Dangerous DelegateCall	17	100%	0	100%	0	N.A.
Integer Overflow	98	100%	0	N.A.	3350	60%
Integer Underflow	224	80%	0	N.A.	2246	60%
Freezing Ether	15	60%	0	N.A.	0	N.A.

percentage of true positives of the identified vulnerabilities. For all categories, sFuzz finds more vulnerable contracts than ContractFuzzer. Note that Contract-Fuzzer removes *Freezing Ether* from their source code and does not check *Integer Overflow/Underflow*. In total, sFuzz finds vulnerabilities in 1113 contracts, i.e., 24 times more than that of ContractFuzzer.

To evaluate the soundness of sFuzz, we manually examine the identified vulnerable contracts to check whether they are true positives or not. However, we are unable to manually check all the identified vulnerability for two reasons. First, there is an overwhelming number of vulnerabilities. Instead, we randomly sample 50 vulnerable contracts with source code in each category and manually check whether the identified vulnerability is a true positive or not. If there are fewer than 50 vulnerable contracts with source code in the category, we check all of them.

For *Gasless Send*, *Exception Disorder* and *Reentrancy* vulnerability, all 50 sampled vulnerable contracts are true positives. For *Time-stamp Dependency*, out of the 50 sampled vulnerable contracts, 43 of them are true positives. In the

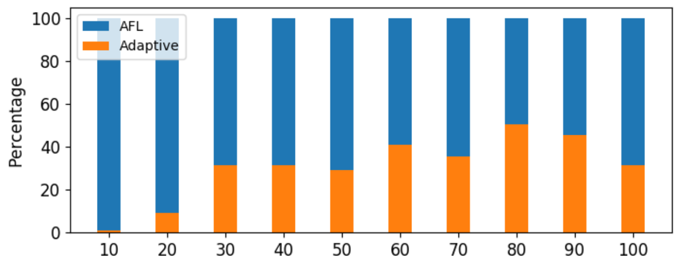


FIGURE 3.9: Percentage of test cases due to adaptive strategy

remaining 7 contracts, although *block.timestamp* and/or *now* is used in a condition, they are irrelevant to the Ether sending part (i.e., no control/data dependency). Rather their values are saved in global variables to record the creation time of specific events. sFuzz mistakenly claims that such cases are vulnerable. For *Block Number Dependency*, 40 out of the 50 sampled vulnerable contracts are true positives. Similarly, the reason for the 10 false positives is the value of *block.number* is assigned to global variables but they are irrelevant to Ether sending process. For *Dangerous DelegateCall*, all 17 sampled contracts are indeed vulnerable. Similarly so for *Integer Overflow*. For *Integer Underflow*, 40 of the 50 identified contracts are indeed vulnerable. The reason for the 10 false positives is because it is non-trivial to identify the correct type of a variable based on bytecode only (e.g., whether it is *uint256* or *uint128*), sFuzz conservatively assumes that all arithmetic operations returning a negative value may be vulnerable. This can be improved by adopting the approach in [29] to infer types based on EVM bytecode. Lastly, for *Freezing Ether*, 9 of the 15 identified contracts are true positives. The reason for the 6 false positives is that although there is a program path which allows the contract to send Ether, the program path is not covered and sFuzz falsely assumes that there is no such program path. This percentage of such false positives is expected to be reduced if sFuzz is applied for a longer time (with more branches covered).

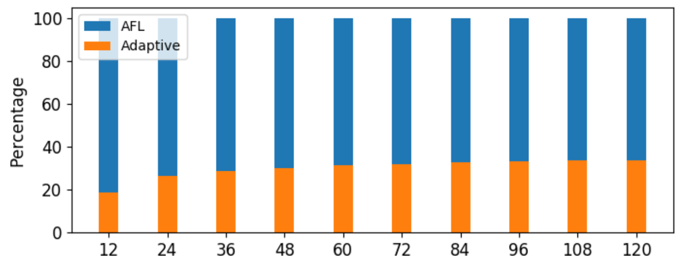


FIGURE 3.10: Effective of adaptive strategy over time

The last column in Table 3.2 shows the results of Oyente. The results should be taken with a grain of salt since Oyente requires the source code. For instance, it is trivial to know the type of variables with the source code, and thus Oyente identifies many more problems with *Integer Overflow/Underflow*. For the remaining vulnerabilities, Oyente does not support 5 of them; identifies a higher number of vulnerable contracts for *Reentrancy* but with a higher false positive rate; and identifies much fewer vulnerable contracts for *Timestamp Dependency*.

3.5.3 Adaptiveness

To answer RQ3, we systematically analyze the test suite generated by sFuzz for each smart contract. Note that each test case covers at least one branch which is not covered by any other test cases. To measure how the two fuzzing strategies implemented in sFuzz complement each other, we count how many test cases in the resultant test suites are generated due to the AFL strategy and how many are due to the adaptive strategy. Note that a test case is judged to be due to the adaptive strategy if and only if it is generated based on a seed selected by line 11 at Algorithm 2.

The results are shown in Figure 3.9, where the y-axis is the percentage of test cases generated by the strategy. Each bar represents 10% of the contracts. We remark that the two strategies have different targets and thus whether they are effective largely depends on what branching conditions are in the smart contracts. We thus sort the contracts according to the speed of sFuzz. The bar on the rightmost thus represents the top 10% contracts. We observe that, as expected, the AFL strategy easily covers most of the branches (since the conditions for executing most branches are not strict). For about 80% of the smart contracts, the adaptive strategy makes a noticeable contribution, i.e., contributing an average of 31% of the generated test cases. Given that sFuzz is applied for each contract only for 2 minutes, the result is encouraging as we hypothesize that the effect of the adaptive strategy would be more apparent if sFuzz is applied for a longer period of time.

To test our hypothesis, we record the percentage of test cases generated by the adaptive strategy every 12 seconds. The results are shown in Figure 3.10, where the x-axis is the fuzzing time and each bar shows the percentage after certain number of seconds. We can observe that the percentage of generated test cases by adaptive strategy increases with more fuzzing time. On average, the percentage rises from 18% after 12 seconds fuzzing to 33% after 2 minutes fuzzing. From the results, we conclude the adaptive strategy is useful in increasing the coverage of the generated test suites.

Threat to validity: There are both internal threats and external threats to our work. For external threats, it is probable that sFuzz's performance will vary with the choice of the initial population, as other researchers have noted [77]. For internal threats, the percentage of true positives in Table 3.2 may not be accurate as they are approximated by a sample of 50 contracts for each type of vulnerability. In addition, the exact intention of the author of the contract is not always clear, even if we try our best to read the source code.

3.6 Related Work and Conclusion

sFuzz is closely related to existing fuzzers for smart contracts. ContractFuzzer [26] is a fuzzer which can check 7 different types of vulnerabilities. Its approach, however, does not use any feedback to improve the test suite. Echidna [82] is another fuzzer that is reportedly capable of checking if the contract violates some user-defined properties. However, we fail to find any publication about it.

sFuzz is complementary to existing symbolic execution engines for smart contracts. Luu *et al.* [23] presented an engine to find potential security bugs in smart contracts. The tool, however, is neither sound nor complete. Krupp and Rossow presented teEther [27], which is focused on financial transactions and related vulnerabilities. Nikolic *et al.* presented a tool named MAIAN [28], which can find 3 types of trace vulnerabilities. Torres *et al.* presented Osiris [29], a tool that combines symbolic execution and taint analysis to discover 3 types of integer bugs in smart contracts. Different from the above works, sFuzz is a fuzzer and it can be combined with the above engines to form a hybrid fuzzing engine.

sFuzz is related to work on formal verification of smart contracts. Zeus [83] is a framework which verifies the correctness and fairness of smart contracts based on LLVM. Bhargavan *et al.* proposed a framework to verify smart contracts formally by transforming the source code and the bytecode to F*, a language designed for verification [84]. Yoichi Hirai presented an attempt to verify the Deed contract using Isabelle/HOL [85].

sFuzz is broadly related to work on analyzing smart contracts. Delmolino *et al.* showed that writing a safe smart contract is not a trivial task [86]. Atzei *et al.*

provided a taxonomy for common vulnerabilities in smart contracts with real-world attacks [87]. M. Fröwis and R. Böhme performed a call graph analysis and showed that only 40% of smart contracts are truthless as their control flows are immutable [88]. Chen *et al.* presented 7 gas-cost programming patterns and showed that most of the contracts suffer from these gas-cost patterns [89].

To conclude, in this work, we present sFuzz, an adaptive fuzzing engine for EVM smart contracts. Experimental results show that sFuzz is significantly more reliable, faster, and more effective than existing fuzzers. sFuzz is currently under rapid development and has already gained interest from multiple companies and research organizations.

Chapter 4

iContract: An Idealist's Approach for Smart Contract Correctness

In this work, we experiment an idealistic approach for smart contract correctness verification and enforcement, based on the assumption that developers are either desired or required to provide a correctness specification due to the importance of smart contracts and the fact that they are immutable after deployment. We design a static verification system with a specification language which supports fully compositional verification (with the help of function specifications, contract invariants, loop invariants and call invariants). Our approach has been implemented in a tool named iContract which automatically proves the correctness of a smart contract statically or checks the unverified part of the specification during runtime. Using iContract, we have verified 10 high-profile smart contracts against manually developed detailed specifications, many of which are beyond the capacity of existing verifiers. Specially, we have uncovered two ERC20 violations in the BNB and QNT contracts.

4.1 Introduction

"After this decade, programming could be regarded as a public, mathematics-based activity of restructuring specifications into programs."

(Edsger W. Dijkstra, 1969)

And it did not happen. Worse yet, the idea of having a formal specification either before or alongside with a program has become unimaginable for ordinary programmers nowadays.

We however may not have the luxury NOT to have a correctness specification when it comes to smart contracts. Smart contracts are programs that run on top of blockchain. They are often used to implement financial applications and increasingly other critical applications. A bug in a smart contract thus could result in a massive loss of valuable digital assets, which has been demonstrated time and time again [73], [90]. More importantly, due to the immutability of blockchain (which is one of its fundamental properties), a smart contract cannot be patched once it is deployed. In other words, once deployed, a bug in the smart contract would make it forever vulnerable. We thus must make sure a smart contract is correct before it is deployed.

Existing approaches on tackling the correctness of smart contracts can be roughly categorized into two groups, i.e., those approaches which target common vulnerabilities and those which support (manually specified) full correctness specification. The former includes an extensive list of approaches and tools on static analysis (such as Mythril [91], Oyente [23] and Securify [24]), fuzzing (such as sFuzz [43], Echidna [82], and ConFuzzius [92]), as well as runtime monitoring (such as sGuard [45], Solythesis [93], and Elysium [34]). While the approaches are different, what is common across these approaches is that they all focus on a collection of generic bugs (such as reentrancy, overflow or underflow, frontrunning and frozen funds). While these approaches are undoubtedly useful, they are incapable of identifying contract-specific bugs or showing their absence.

In this work, we propose iContract, a fully compositional verification system for verifying and enforcing the correctness of smart contracts. iContract supports a rich specification language which allows developers to specify not only the traditional loop invariants and function specifications but also contract invariants (for contract-level specification) and call invariants (for specification of external function calls). We remark that designing a specification language that is relatively easy to use (which is essential in practice), expressive, and makes verification easy is nontrivial. For instance, a smart contract often interacts with other contracts via interfaces. Mishandling such interfaces (e.g., assuming that no contract states are modified by such interfaces or contracts states can be modified arbitrarily) would hinder the verification of contracts. In this work, we annotate external function calls with call invariants (so that we can quantify the behavior of the external function call using a correctness logic formula as well as an incorrectness logic formula). These call invariants can be validated at the runtime and relied upon as assumptions when we verify the calling function.

To evaluate the effectiveness and applicability of iContract in practice, we apply iContract to verify 10 real-world high-profile contracts. For each contract, a full specification of its correctness is first developed manually, with a total of 1 PhD-month. iContract is then applied to verify each of the contracts. The results show that iContract not only is scalable for verifying real-world contracts but also uncovering contract-specific bugs. The results are encouraging as it shows that developing a specification for critical but relatively simple programs such as smart contracts is entirely feasible.

To sum up, our main contributions are as follows. First, we propose an approach for the correctness specification of smart contracts which facilitate completely compositional verification, including revert specification (i.e., specifications that capture explicit reverts) as well as call invariants for frame conditions. Second, we develop an implementation of the compositional verification approach for real-world Solidity smart contracts. Lastly, we conduct an evaluation using 10 real-world high-profile smart contracts (with a full specification of their correctness).

4.2 Overview

4.2.1 Vulnerability and Correctness

Same as traditional programs, smart contracts can have bugs. For instance, a long list of common bugs have been identified [94], some of which have been exploited and huge financial losses have occurred [73]. Making sure that a smart contract does not repeat the same mistakes merely constitutes the first step towards contract correctness.

An ideal approach for smart contract correctness verification must satisfy the following requirements. First, it must support a rich notion of correctness. This is because each contract is designed for a unique purpose and thus is expected to satisfy a contract-specific specification. Existing approaches that are designed to verify smart contracts against common general vulnerabilities are thus insufficient. Second, it must be fully compositional, i.e., given a contract, we should be able to establish its correctness without relying on external contracts. Furthermore, each functional unit, such as a function or even a loop, should have its own specification so that any kind of global reasoning (even at the contract level) could be avoided. In so doing, the verification system could achieve scalability. Third, it must be fully automatic once the specification is provided. Lastly, it must guarantee that the smart contract satisfies its specification, either through static verification (ideally) or runtime verification (if necessary).

We obviously must pay some price to achieve the above-mentioned goals. Our approach is thus based on two assumptions. First, we make the strong assumption that developers are either requested or required (by stakeholders or certification boards) to provide a correctness specification. While it was sadly proven too strong an assumption for ordinary programs, it may be justifiable for smart contracts due to the reasons mentioned above. Second, we make the assumption that developers are willing to pay some reasonable amount of additional fee (i.e., for runtime checking) in order to guarantee that the smart contract satisfies the specification.

4.2.2 An Illustrative Example

In the following, we illustrate how our goals are achieved by iContract through an example. Figure 4.1 shows a token-issuing smart contract (written according to the ERC20 standard [95]), which is a simplified version of a real-world smart contract named HEALTH[1]. The contract includes global variables $burnFee$, $devFee$, $bFee$, $uniswapV2$, and $balances$. It supports (through a public function) *transfer*

[1]deployed at BNB chain address 0x32b166e082993af6598a89397e82e123ca44e74e

```
 1 contract Health {
 2   ...
 3   /// reverts_if(_balances[from] < value)
 4   /// ensures(to != uniswapV2 && value == 0 && _balances[from] >= value, _balances
         [_burnAddress] == old(_balances[_burnAddress]))
 5   function _transfer(address from, address to, uint value) private {
 6     require(_balances[from] >= value);
 7     // require(value > 0);
 8     if (to == uniswapV2) {
 9       UniswapRouter(uniswapV2).swapAndLiquify(numTokensSell);
10       /// call_inv(_balances[this] >= numTokensSell, _balances[this] == old(
             _balances[this]) - numTokensSell && _balances[uniswapV2] == old(
             _balances[uniswapV2]) + numTokensSell)
11       /// call_modifies(_balances[this], _balances[uniswapV2])
12     }
13     if (from != uniswapV2) {
14       uint burnValue = _balances[uniswapV2].mul(burnFee).div(1000);
15       _balances[uniswapV2] = _balances[uniswapV2].sub(burnValue);
16       _balances[_burnAddress] = _balances[_burnAddress].add(burnValue);
17       IPancakePair(uniswapV2).sync();
18       /// call_modifies()
19     }
20     uint devValue = value.mul(devFee).div(1000);
21     uint bValue = value.mul(bFee).div(1000);
22     uint newValue = value.sub(devValue).sub(bValue);
23     _balances[from] = _balances[from].sub(value);
24     _balances[to] = _balances[to].add(newValue);
25     _balances[address(this)] = _balances[address(this)].add(devValue);
26     _balances[_burnAddress] = _balances[_burnAddress].add(bValue);
27   }
28
29   function transfer(address to, uint value) public returns(bool) {
30     _transfer(msg.sender, to, value);
31     return true;
32   }
33 }
```

FIGURE 4.1: A sample contract

of HEALTH tokens (hereafter h-tokens) from account *from* (a.k.a. sender) to account *to* (a.k.a. receiver). Note that the sender is charged with some fee for the transfer. Furthermore, in some cases, it burns (subtracts) an amount (proportional to *value*) of the h-tokens hold by $uniswapV2$, which is a service that swaps h-tokens with BNB (i.e., a token which is often used for token exchange services) or vice versa. Particularly, first, at lines 8–12 if the receiver is $uniswapV2$, the contract swaps $numTokensSell$ h-tokens for BNB (line 9). Second, at lines 13–19, if the sender is not $uniswapV2$, the contract burns some h-tokens from $uniswapV2$ (line 15). Lastly, at lines 20–26, the contract charges development fee (line 25), burns token (line 26), and transfers the remaining (line 24) to receiver.

To verify the contract, we start with developing a correctness specification. For instance, lines 3–4, 10–11 and 18 constitute the correctness specification of the function *_transfer*. The specification relies on a set of pre-defined functions, such as reverts_if(p), modifies($\overline{x}$), ensures(p, q), call_modifies($\overline{x}$) and call_inv(p, q). Intuitively, reverts_if(p) says that the transaction reverts if p is satisfied; modifies($\overline{x}$) (respectively call_modifies($\overline{x}$)) says that the function (respectively the external call) only modifies those variables in $\overline{x}$; ensures(p, q) is equivalent to the Hoare triple $\{p\}s\{q\}$ where s is the function body; and call_inv(p, q) right after a function call is a call invariant, where p is a precondition of the call and q is expected to be satisfied after the call. We remark that modifies($\overline{x}$), call_modifies($\overline{x}$) can be regarded as syntactic sugars of certain special cases of ensures(p, q) and call_inv(p, q).

In particular, the specification at line 4 demands that when $value = 0$, no token should be burned. This is important as burning h-tokens reduces the total supply and, thus, increases the price of h-tokens. If h-tokens can be burned unintentionally (e.g., when $value = 0$), attackers could potentially use the function to manipulate the market price. According to the call_modifies($\overline{x}$) at line 11, only variables $\_balances[this]$ and $\_balances[uniswapV2]$ are modified. The call invariants at lines 10–11 state that the function call at line 9 transfers $numTokensSell$ h-tokens from address $this$ to address $uniswapV2$. In particular, the $balances[this]$ is reduced and $balances[uniswapV2]$ is increased by the same amount. By default, all global variables could be modified in the called function. Line 18 specifies that no variables are modified by the external call.

Once the specification is given, iContract systematically verifies the contract against the specification. It reports that the specification at line 4 is falsified with a counterexample, i.e., if the sender is not $uniswapV2$ and *value* is 0, h-tokens are burned from $uniswapV2$ on line 15. In other words, this contract could be exploited by abusing the function *_transfer* to burn h-tokens and manipulate its price, i.e., an attacker first buys some h-tokens, repeatedly calls *_transfer* as described above, and sells his h-tokens at a higher price.

With the verification result, we can prevent the manipulation by adding one statement $require(value > 0)$ at line 7. Afterwards, iContract reports that the specification is successfully verified. This is because if $value = 0$, the function is reverted. Furthremore, if the user wish to verify the revert, he could annotate another specification as reverts_if(value=0) and invoke iContract to verify it. Indeed, our system could verify the revert scenario successfully. Alternatively, if the user chooses to conduct runtime verification, iContract automatically translates the above-mentioned unverified specification into an assertion, which is then validated every time the function is invoked. Note that in the latter case, additional gas will be paid (for executing the assertion) for the correctness.

4.3 Specification Language

4.3.1 High-level Overview

In the following, we present our specification language which is designed to support fully compositional verification of smart contracts at the function level. At a high-level, our specification is composed of function specifications, loop invariants, (external) call invariants and contract invariants.

Function specifications: Ideally, a user would be able to read the function specification and be fully aware of what the function does. Given a function f, a function specification takes the form of multiple ensures(p, q) statements (at the beginning of the function body), where p and q are predicates that we shall define shortly. Each ensures(p, q) statement represents a Hoare triple $\{p\}f(\overline{x})\{q\}$, i.e., any reachable state at the end of the function (i.e., without reverting) from a state satisfying p must satisfy q. In other words, q is an over-approximation of the states reachable from p.

Loop invariants: It is well known that loops are difficult when they come to program verification. While there are many existing approaches on synthesizing loop invariants [96], [97], for now, we make the assumption that loop invariants are provided as a part of the specification. A loop invariant takes the form of multiple loop_inv(q) statements at the beginning of the loop. Given a loop $while\ b\ do\ s$, loop_inv(q) at the beginning of the loop represents a Hoare triple $\{b \wedge q\}s\{\neg b \wedge q\}$.

Call invariants: Smart contracts often rely on other smart contracts through external function calls. To avoid global analysis, we assume that each external call is associated with a specification in the form of multiple call_inv(p, q) statements and multiple achieves(p, q) statements. These help to ensure the function call behaves expectedly, i.e., they serve as the minimal requirements on the external contracts that are needed to guarantee the correctness of this contract. Given a function call $m(\overline{e})$, a statement call_inv(p, q) forms a triple $\{p\}m(\overline{e})\{q\}$. If p is satisfied before the call, q is always satisfied after the execution of the function call. Such statements can be used to prevent the well-known reentrancy vulnerability. A statement achieves(p, q) forms a specification in the incorrectness logic [98], which intuitively means that if p is satisfied, it is possible to satisfy q by making the external call.

Contract invariants: A contract invariant takes the form of multiple cinv(p) statements at the top of the contract and is expected to be satisfied after executing the constructor and every public function in the contract. Although technically it can be captured using function specifications (for both the constructor and every public function), it is typically used to capture contract-level behaviors that are expected to hold always regardless of the functionalities provided in the contract.

In addition, iContract supports a number of syntactic sugars which ease the writing of specification. For instance, for each function, loop, or external function call, we assume that all global variables may be modified unless a modifies($\overline{x}$) statement is put in place (e.g., function definitions, function calls), which specifies that all except those variables in $\overline{x}$ remain unchanged. Additionally, when variable x is a mapping, we allow users to write modifies($x[a]$) where a is constant value to state that only the value at location a of x is modified, while the values at other locations are not.

In terms of specifying the expected behaviors of smart contracts, our specification language has mulitple advantages over existing approaches [30]–[32]. First, our specification language is designed to avoid global reasoning with the help of call invariants. Second, the reverts_if(p) statements allow us to easily capture explicit reverts which are very common in smart contracts in the form of $require$, $revert()$ and so on. Note that this feature is missed from approaches such as Solc-verify and as a result, those respective tools often generate false alarms, i.e., reporting violation of postcondition on transactions that ought to be reverted. Last, our specification is mostly based on well-known and well-founded concepts which makes it easy to adopt.

TABLE 4.1: Core features of Solidity

Func m	$m(\overline{v}) = s$
Stmt s	s_A | $s; s$ | if e then s else s | while e do s | require(p) | assert(p) | skip
Atom s_A	$v := e$ | $v.m := e$ | $v[e] := e$
Expr e	l | v | $v.m$ | $v[e]$ | $e \oplus e$ | $\odot e$ | $m(\overline{e})$

4.3.2 Formalization

In the following, we provide the necessary formalization of our specification language as well as smart contracts so that we can present precisely how our approach works. Note that since all our verification effort (including static verification and runtime verification) takes place at the function-level, all we need to formalize are smart contract functions and function-level specification.

Defining smart contracts: To ease the discussion hereafter, we model Solidity's core (function-level) features using the language presented in Table 4.1. A function m includes parameters $\overline{v}$, and a body statement s. A statement s is an atomic statement s_A, a conditional statement, a while loop, an assertion, revert statement, and it also can be a sequence of statements (according to the definition shown in Table 1). An atomic statement s_A is an assignment to a variable ($v := e$), an assignment to member of a variable ($v.m := e$), or an assignment to an array element ($v[e] := e$). An expression e is a literal l, a variable v, a member access $v.m$, an index access $v[e]$, a binary expression $e \oplus e$, a unary expression $\odot$ e, or a call $v.m(\overline{e})$ of a local function (in the same contract) or an external function (in a different contract). We use *rev* as a preserved variable for revert condition: It is true if the contract has been reverted. Note that we can simply transform other Solidity features into our core language features such as the statement $require(a)$ is equivalent to the statement i) $assert(a \wedge \neg rev)$ in verifying code against a function variant or ii) $revert(\neg a \wedge rev)$ in the verification of reverts_if(...).

To define the semantics of smart contracts, we define a set Var contains all the variables in the contract, a set Mem contains all the members of the data structures in the contract, a set of mapping for arrays A, and data structures (where $A \cap Var = \emptyset$), a set Loc contains all the memory locations, a set Val contains all non-memory values (i.e., $Val = Int \cup Float \cup Bool \cup Str$, with Int, $Float$, $Bool$, and Str are the sets containing integer, floating-point, boolean, and string literals). We use two mapping functions $S \in Stacks$ and $H \in Heaps$ to keep track of the execution environment. Consequently, a program state $\sigma_c \in States$ is defined by a pair of stack and heap, as follows.

$$\begin{aligned} S \in Stacks &=_{\text{def}} Var \to (Val \cup Loc) \\ H \in Heaps &=_{\text{def}} Loc \to (Type \to (Mem \cup Int) \to (Val \cup Loc)) \\ \sigma_c \in States &=_{\text{def}} Stacks \times Heaps \end{aligned}$$

where the set $Type$ contains all the data structure types defined in the contract as well as the array type.

We define a standard small-step operational semantics of smart contracts (based on the semantics of Solidity). A configuration C is a pair (s, σ_c) where s is a program and σ_c is a program state (i.e., the valuation of both S and H). The semantics is given by a binary relation, $\rightsquigarrow$, on configurations. Its intended interpretation is that $(s, \sigma_c) \rightsquigarrow (s', \sigma_c')$ holds if the execution of the statement in the configuration (s, σ_c) can result in the new program configuration (s', σ_c'). An execution (of s) is a possibly infinite sequence of configurations $(C_i)_{i \geq 0}$ with

$C_0 = (s, \_)$ such that $C_i \leadsto C_{i+1}$ for all $i \geq 0$. We define $\leadsto^*$, the reflexive-transitive closure of $\leadsto$, to capture finite executions $(C_i)_{0 \leq i \leq n}$. The details of the small step semantics is present in Figure 4.2.

$$\frac{}{\langle S, H\rangle \vdash l \Downarrow l}\ \textsc{Const} \qquad \frac{S(v) = l \quad H(l) = (type(v), m, k)}{\langle S, H\rangle \vdash v.m \Downarrow k}\ \textsc{Access}$$

$$\frac{}{\langle S, H\rangle \vdash v \Downarrow S(v)}\ \textsc{Var} \qquad \frac{S(v) = l \quad S(e) = i \quad H(l) = (type(v), i, k)}{\langle S, H\rangle \vdash v[e] \Downarrow k}\ \textsc{Select}$$

$$\frac{\langle S, H\rangle \vdash e_1 \Downarrow k_1 \quad \langle S, H\rangle \vdash e_2 \Downarrow k_2}{\langle S, H\rangle \vdash e_1 \oplus e_2 \Downarrow k_1 \oplus k_2}\ \textsc{Binary} \qquad \frac{\langle S, H\rangle \vdash e \Downarrow k_1}{\langle S, H\rangle \vdash \odot e \Downarrow \odot k_1}\ \textsc{Unary}$$

$$\frac{}{\langle S, H\rangle, \mathsf{revert}; s_2 \leadsto \langle S_0, H_0\rangle, \mathsf{skip}}\ \textsc{Revert} \qquad \frac{}{\langle S, H\rangle, \mathsf{skip}; s_2 \leadsto \langle S, H\rangle, s_2}\ \textsc{Skip}$$

$$\frac{\langle S, H\rangle, s_1 \leadsto \langle S_1, H_1\rangle, s_{1'}}{\langle S, H\rangle, s_1; s_2 \leadsto \langle S_1, H_1\rangle, s_{1'}; s_2}\ \textsc{Seq} \qquad \frac{\langle S, H\rangle, s_1 \leadsto \langle S_1, H_1\rangle, \mathsf{abort}}{\langle S, H\rangle, s_1; s_2 \leadsto \langle S_1, H_1\rangle, \mathsf{abort}}\ \textsc{Seq-Err}$$

$$\frac{\langle S, H\rangle \vdash e \Downarrow k \quad S_1 = S[v \leftarrow k]}{\langle S, H\rangle, v := e \leadsto \langle S_1, H\rangle, \mathsf{skip}}\ \textsc{Assign-1} \qquad \frac{\langle S, H\rangle \vdash v \Downarrow k \quad k \notin dom(H)}{\langle S, H\rangle, v.m := e \leadsto \langle S, H\rangle, \mathsf{abort}}\ \textsc{Err1}$$

$$\frac{\begin{array}{c}\langle S, H\rangle \vdash v \Downarrow k \quad k \in dom(H) \quad \langle S, H\rangle \vdash e \Downarrow k_1 \\ H_1 = H[(k, type(v), m) \leftarrow k_1]\end{array}}{\langle S, H\rangle, v.m := e \leadsto \langle S, H_1\rangle, \mathsf{skip}}\ \textsc{Assign-2}$$

$$\frac{\begin{array}{c}\langle S, H\rangle \vdash v \Downarrow k \quad \langle S, H\rangle \vdash e_1 \Downarrow k_1 \quad \langle S, H\rangle \vdash e_2 \Downarrow k_2 \\ H_1 = H[(k, Array, k_1) \leftarrow k_2]\end{array}}{\langle S, H\rangle, v[e_1] := e_2 \leadsto \langle S, H_1\rangle, \mathsf{skip}}\ \textsc{Assign-3}$$

$$\frac{\langle S, H\rangle \vdash v \Downarrow k \quad k \notin dom(H)}{\langle S, H\rangle, v[e_1] := e_2 \leadsto \langle S, H_1\rangle, \mathsf{abort}}\ \textsc{Err2} \qquad \frac{\langle S, H\rangle \vdash e_1 \Downarrow k_1 \quad k_1 \notin size(v)}{\langle S, H\rangle, v[e_1] := e_2 \leadsto \langle S, H_1\rangle, \mathsf{abort}}\ \textsc{Err3}$$

$$\frac{\langle S, H\rangle \vdash b \Downarrow True}{\langle S, H\rangle, \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s' \leadsto \langle S, H\rangle, s}\ \textsc{If-T} \qquad \frac{\langle S, H\rangle \vdash b \Downarrow False}{\langle S, H\rangle, \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s' \leadsto \langle S, H\rangle, s'}\ \textsc{If-F}$$

$$\frac{\langle S, H\rangle \vdash b \Downarrow True}{\langle S, H\rangle, \mathsf{while}\ b\ \mathsf{do}\ s \leadsto \langle S, H\rangle, s; \mathsf{while}\ b\ \mathsf{do}\ s}\ \textsc{Loop-T}$$

$$\frac{\langle S, H\rangle \vdash b \Downarrow False}{\langle S, H\rangle, \mathsf{while}\ b\ \mathsf{do}\ s \leadsto \langle S, H\rangle, \mathsf{skip}}\ \textsc{Loop-F}$$

$$\frac{v.m(\overline{p}) = s \quad \langle S, H\rangle \vdash \overline{e} \Downarrow \overline{k} \quad S' = S[\overline{p} \leftarrow \overline{k}]}{\langle S, H\rangle \vdash v.m(\overline{e}) \leadsto \langle S', H\rangle, s}\ \textsc{Call}$$

FIGURE 4.2: Small-step operational semantics of the smart contract language, given by the binary relation $\leadsto$ over $Stacks \times Heaps$

$$
\begin{array}{lcllll}
S,H & \models & \Phi_1 \vee \Phi_2 & \textbf{iff} & (S,H \models \Phi_1) \vee (S,H \models \Phi_2) \\
S,H & \models & \Psi_1 \wedge \Psi_2 & \textbf{iff} & (S,H \models \Psi_1) \wedge (S,H \models \Psi_2) \\
S,H & \models & a_1 \otimes a_2 & \textbf{iff} & (S,H \models a_1 = k_1) \wedge (S,H \models a_2 = k_2) \wedge (k_1 \otimes k_2) \\
S,H & \models & a_1 \oplus a_2 = k & \textbf{iff} & (S,H \models a_1 = k_1) \wedge (S,H \models a_2 = k_2) \wedge (k_1 \oplus k_2 = k) \\
S,H & \models & \odot a = k & \textbf{iff} & (S,H \models a = k_1) \wedge (\odot k_1 = k) \\
S,H & \models & l = l & \textbf{iff} & \textbf{true} \\
S,H & \models & v = k & \textbf{iff} & S(v) = k \\
S,H & \models & v[a] = k & \textbf{iff} & S(v) \in dom(H) \wedge 0 \geq S(a) < size(v) \wedge H(S(v)) = (Array, S(a), k) \\
S,H & \models & v.m = k & \textbf{iff} & S(v) \in dom(H) \wedge H(S(v)) = (type(v), m, k) \\
S,H & \models & old(v) = k & \textbf{iff} & S_0, H_0 \models v = k \\
S,H & \models & sum(v) = k & \textbf{iff} & type(v) = Array \wedge S(v) \in dom(H) \wedge \sum_{i=0}^{size(v)} \{v[i]\} = k
\end{array}
$$

FIGURE 4.3: Specification formula semantic where $dom(f)$ returns the domain of function f, $size(v)$ the range of index of the array v.

Defining the specification language: Our specification language is constituted of predicates defined using the syntax below.

$$
\begin{array}{rclrcl}
\Phi, p, q & := & \Psi \mid \Phi \vee \Phi & \Psi & := & a \otimes a \mid \Psi \wedge \Psi \\
a & := & e \mid a \oplus a \mid \odot a & e & := & l \mid v \mid v[a] \mid v.m \mid old(v) \mid g(v)
\end{array}
$$

In general, a predicate Φ is a disjunction with one or multiple conjunctions Ψ. Each conjunct in Ψ is a relational predicate with $\otimes$ is a relational operator (i.e., $>$, $\geq$, $=$, $\neq$, $<$, $\leq$). The left-hand side and right-hand side of a relational predicate are arithmetic expressions. An arithmetic expression may have one atomic expression or multiple of them connected by binary operators $\oplus$ (i.e., $+$, $-$, $*$, $/$) or unary operators $\odot$ (i.e., $\neg$, $-$). An atomic expressions includes a literal l, a variable v, a member access $v.m$, and an index access $v[a]$. The expression $v.m$ accesses the value stored in the member m of a struct v, whereas the expression $v[a]$ accesses the value at key a of a mapping v. In addition, we provide a function $old(v)$ which returns the value of variable v at the beginning of the function (for function specifications) or the loop (for loop invariant) or before an external function call (for call invariants). Moreover, we support a library of externally defined function $g(v)$. One example is the sum function, which, given a mapping v, computes the sum of all values stored in v.

The semantics is defined according to a satisfaction relation $S, H \models \Phi$ which is defined in a common way, as shown in Figure 4.3. Next, we define the correctness in our specification language. First, regarding contract invariants, given a contract c associated with multiple cinv(p) statements, the contract is correct if and only if each ensures(p, p) is satisfied by all the public functions including the constructor. Second, regarding function specifications, given a function $m(\overline{v}) = s$ associated with multiple ensures(p, q) and reverts_if(p′) statements, the function is correct iff for each ensures(p, q) statement, the following is satisfied.

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma_c') \implies \sigma_c' \models q$$

Furthermore, for each reverts_if(p′) statement, the following is satisfied

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models p' \wedge (s, \sigma_c) \rightsquigarrow^* (\text{require(b)}, \sigma_c') \implies \sigma_c' \models \neg b$$

Third, regarding loop invariants, given a loop $while\ b\ do\ s$ associated with an loop_inv(q) statement at the beginning, the following must be satisfied where L is a function that filters states satisfying b.

$$\forall \sigma_c, \sigma_c'.\ \sigma_c \models q \wedge (s, L(\sigma_c, b)) \rightsquigarrow^* (\text{skip}, \sigma_c') \implies \sigma_c' \models q$$

Fourth, for each achieves(p, q), the following must be satisfied.

$$\forall \sigma_c'.\exists \sigma_c.\ \sigma_c' \models q \implies \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma_c')$$

Lastly, regarding call invariants, given an external function call $m(\overline{e})$ associated with multiple call_inv(p, q), for any implementation s of $m(\overline{e})$, the following must be satisfied: $\forall \sigma_c, \sigma'_c.\ \sigma_c \models p \wedge (s, \sigma_c) \rightsquigarrow^* (\text{skip}, \sigma'_c) \implies \sigma'_c \models q$.

4.4 Verification

We present our compositional verification algorithm by first defining an encoding function $post(\sigma_i, s_i)$ and illustrating how to utilize it to validate function specification ensures(p, q) and revert specification reverts_if(p). We remark that we abuse the notation σ_i to represent a symbolic state where its syntax is similar to our specification language. Furthermore, we provide encoding rules that substitute loops and function calls with their specifications.

4.4.1 Function validation

We define an encoding function $post(\sigma_i, s_i)$ that takes a pre-state σ_i and a statement s_i as inputs, and procedure post-states σ_k as output. Given a function $m(\overline{v}) = s$ which may contain loops as well as internal and external function calls, our validations are defined as follows. A function specification ensures(p, q) with implementation s is verified if $post(p, s)$ returns σ such that $\sigma \Rightarrow q$. The execution $post(p, s)$ indicates that the encoding process starts with pre-state p. After processing the statement s, the validation formula $\sigma \Rightarrow q$ means that if the function is not reverted then the encoding starts with p implies the post-condition q. Similarly, a specification reverts_if(p) is verified if $post(p, s)$ returns the post-state $\sigma \wedge$ *rev* at exits. Note that the procedure of verifying ensures(p, p) utilizes the encoding rule REVERT-1. On the other hand, other REVERT rules, such as REVERT-2 and REVERT-3, are employed for the verification of reverts_if(p).

$$\frac{\sigma, s_1 \rightsquigarrow q_1 \qquad q_1, s_2 \rightsquigarrow \sigma_2}{\sigma, s_1; s_2 \rightsquigarrow \sigma_2}\ \textsc{Seq} \qquad \frac{\sigma' = \exists x.\, \sigma[x/v] \wedge v = e[x/v]}{\sigma, v := e \rightsquigarrow \sigma'}\ \textsc{Assign-1}$$

$$\frac{u' = u + e - v[m] \qquad \sigma' = \exists x.\, \sigma[x/v] \wedge v = x[m[x/v] \to e[x/v]]}{\sigma \wedge \text{sum}(\text{v}) = u, v[m] := e \rightsquigarrow \sigma' \wedge \text{sum}(\text{v}) = u'}\ \textsc{Assign-2}$$

$$\frac{}{\sigma \wedge rev, s \rightsquigarrow \sigma \wedge rev}\ \textsc{Rev-Prop} \qquad \frac{\sigma' = \exists x.\, \sigma[x/v] \wedge v = x[m \to e[x/v]]}{\sigma, v.m := e \rightsquigarrow \sigma'}\ \textsc{Assign-3}$$

$$\frac{\sigma \wedge b, s_0 \rightsquigarrow \sigma_1 \qquad \sigma \wedge \neg b, s_1 \rightsquigarrow \sigma_2}{\sigma, \text{if } b \text{ then } s_0 \text{ else } s_1 \rightsquigarrow \sigma_1 \vee \sigma_2}\ \textsc{If} \qquad \frac{\sigma \Rightarrow q}{\sigma, \text{assert}(\text{q}) \rightsquigarrow \sigma}\ \textsc{Assert}$$

$$\frac{\sigma' = \sigma \wedge p}{\sigma, require(p) \rightsquigarrow \sigma'}\ \textsc{Revert-1} \qquad \frac{\sigma \Rightarrow \neg p}{\sigma, require(p) \rightsquigarrow \sigma \wedge rev}\ \textsc{Revert-2}$$

$$\frac{\sigma \not\Rightarrow \neg p}{\sigma, require(p) \rightsquigarrow \sigma}\ \textsc{Revert-3}$$

$$\frac{\sigma' = \exists \overline{x}.\, \sigma[\overline{x}/\overline{v}] \wedge p \wedge \neg b}{\sigma, \text{modifies}(\overline{v}); \text{loop\_inv}(\text{p}); \text{while } b \text{ do } s \rightsquigarrow \sigma'}\ \textsc{Loop}$$

$$\frac{\text{reverts\_if(p)} \in \textsc{spec}(m(\overline{e})) \qquad \sigma, require(\neg p); m(\overline{e}) \rightsquigarrow \sigma'}{\sigma, m(\overline{e}) \rightsquigarrow \sigma'}\ \textsc{Revert-Inter}$$

$$\frac{\sigma \Rightarrow p \qquad \sigma' = \exists \overline{x}.\, \sigma[\overline{x}/\overline{v}] \wedge q}{\sigma, \text{modifies}(\overline{v}); finv(p, q); m(\overline{e}) \rightsquigarrow \sigma'}\ \textsc{Call-Spec}$$

FIGURE 4.4: Encoding rules (where $finv(p, q)$ is ensures(p, q) or call_inv(p, q))

4.4.2 Generating Proof Obligations

We define encoding function $post(\sigma_i, s_i)$ using encoding rules, each of which is of the following form.

$$\frac{premise_0 \quad ... \quad premise_i}{\sigma_i, s_i \rightsquigarrow \sigma_k}$$

This transition rule means given a pre-state σ_i, a statement s_i, it executes $premise_0$, ..., $premise_i$ to obtain the post-state σ_k. The encoding rules are shown in Figure 4.4. Note that the encoding transforms the code into the predicates supported by off-the-shelf SMT solver Z3. While most of the syntax is self-explanatory, we use the notation $v[a \to l]$ to represent an array with $v[a \to l][a'] = v[a']$ when $a' \neq a$ and $v[a \to l][a] = l$.

The rules are divided into three groups, i.e., rules for local operations, rules for external function calls, and rules for revert. Rules for local operations include SEQ, ASSIGN-1, ASSIGN-2, ASSIGN-3, IF and LOOP. They are similar to the traditional Hoare rules. In the ASSIGN-2 and ASSIGN-3, we substitute v before the assignment with x, and set the current v as the result of update value $e[x/v]$ to the value located at index $m[x/v]$ or property m. In the ASSIGN-2, for each write operation to $v[m]$, we compute sum(v) by adding the current sum u to the difference between the new value e and the old value $v[m]$, i.e, $u' = u + e - v[m]$. The LOOP substitutes the loop with its invariant and exiting condition (i.e., $\neg b$).

Rules for revert include REVERT-1 (the non-revert condition is part of the pre-condition), REVERT-2 (the revert condition met) and REVERT-3 (the revert condition is not met). The idea is that the function is reverted if any of the condition leading to revert is satisfied. If the revert condition is satisfied, the value of *rev* is set, and after that our system skips all the remaining statements by using rule REV-PROP.

Rules for external function calls include CALL-SPEC, which replaces function calls using either function specifications (if it calls for a local function) or call invariants (if it calls for an external function). This rule updates modified variables $\overline{v}$ through the substitutions $\sigma[\overline{x}/\overline{v}]$. Note that, to propagate the reverts_if(p) back to the caller, via rule REVERT-INTER, we simply convert it to $require(\neg p)$ before the function call is encoded. Moreover, each ensures(p, q) is lifted to the context of the current function by substituting free variables appearing on parameters with their corresponding arguments.

Note that the correctness specification may be over-approximating and thus our verification may lead to false alarms and spurious counterexamples. Instead of running test cases with extra costs, the incorrectness specification associated with external function calls is used to construct counterexamples. According to the concrete values from counterexamples, we first determine an execution path leading to the violation, and then use the achieves(p, q) statements associated with the involved external function calls to check whether the counterexample is real. We develop another predicate $post_U(p, s)$ to compute under-approximating post-states for the implementation s, then our system confirms the bug described in the spec if $q \Rightarrow \sigma$. In term of encoding for $post_U$, dropping execution paths is allowed in incorrectness logic. Therefore, the number of loop iterations can be freely chosen. Only the true-branch or the false-branch is selected while encoding an if-statement. If there is an execution path that satisfies the incorrectness specification then the counterexample is determined to be a real violation. Finally, to handle function call $m(\overline{e})$; modifies($\overline{v}$); achieves(p′, q′) at the calling states σ, it first tests $p' \Rightarrow \sigma$. If this test succeeds, it produces $\sigma[\overline{x}/\overline{v}] \wedge q'$ as the post-states. Otherwise, if $\overline{v} \cap \textsc{FreeVars}(\sigma) = \emptyset$, it checks $\textsc{Sat}(p' \wedge \sigma)$. If it is satisfied, it produces $\sigma[\overline{x}/\overline{v}] \wedge q'$ as the post-states. The soundness of the former comes from CONSEQUENCE rule and the later is from CONSTANCY rule in incorrectness logic.

4.5 Implementation and Evaluation

4.5.1 Implementation

iContract is implemented with around 1K lines of Python code. It supports most features of Solidity version 0.5.1 including inheritance and important built-in functions (e.g., *send*, and *call*). iContract uses a locally installed Solidity compiler to compile a user-provided Solidity file into a JSON file containing the typed abstract syntax tree (AST). Then, iContract analyzes the AST to encode contracts into predicates using the Z3 library. We leverage NatSpec [99] format to define our own specifications.

The encoding is mostly straightforward except some relevant details that we discuss below. We use SMT Integer to model int/unsigned and int/address and so on[2]. To support contract inheritance, we implement a symbol table which allows us to query global variables and functions of parent contracts using inheritance tree provided by the Solidity compiler. We also take into account function overriding and variable hiding.

Our current implementation has several limitations. First, it does not support low-level API calls including inline assembly, Application Binary Interface functions, and bitwise operations. Second, iContract does not compute gas consumption to determine out-of-gas exceptions. Last, iContract analyzes contracts without the presence of aliasing. Note that although Solidity allows two variables reference to the same data location (i.e., aliasing), it is not very common in Solidity and we leave it to future work.

4.5.2 Experimental Evaluation

In the following, we design and conduct multiple experiments to answer the following research questions (RQ).

- *RQ1: Can iContract verify real-world smart contracts?*
- *RQ2: How does iContract compare with Solc-verify [32], a state-of-the-art tool for verifying function-level properties?*

[2]Note that runtime checking for arithmetic overflow has been introduced since Solidity 0.8 and thus no longer an issue.

TABLE 4.2: Statistics on verified contracts

Project	#Contracts	#Functions	#Ifs	#Specifications	LOC	#Transactions (mil)
BAT	4	16	20	17	179	3.97
BNB	2	13	22	25	150	1.00
HT	4	13	4	2	127	0.67
HOT	3	22	29	28	279	0.95
IOTX	8	32	28	35	500	0.28
QNT	5	24	13	16	239	1.21
MANA	11	28	21	70	282	2.50
ZIL	9	35	42	70	353	0.44
NXM	3	37	36	40	448	0.12
SHIB	4	33	12	33	448	9.5

RQ1 aims to evaluate whether iContract is useful for some practical smart contracts. RQ2 aims to evaluate whether iContract's approach (in particular, its specification language) can achieve its goals better than existing approaches.

In the following, we present the evaluation results and answer the questions. All our experiments are conducted on a single processor running an Ubuntu 16.04.6 LTS machine with Intel(R) Core(TM) i9-9900 CPU @ 3.10GHz and 64GB of memory. The timeout is set to be 5 minutes for verifying the specification. Our implementation and the verified contracts are available online [100].

RQ1: To answer this question, we identify a set of 10 high-profile projects from EtherScan [80]. The relevant statistic of these contracts is shown in Table 4.2. The table shows the name of each project. For each project, it shows the number of contracts (#Contracts), the number of functions (#Functions), the number of if-statements (#Ifs), the number of specification statements (#Specifications), line of codes (LOC), and the number of transactions (#Transactions) in millions. Most of them have over 200 lines of code and 20 functions. Each project is associated with a Solidity file, which typically contains multiple contracts including a main one as well as library or parent contracts. Since not all smart contracts are written in the same Solidity versions, we have to convert them to a fixed version (i.e., 0.5.1). This is necessary to ensure the consistency of our verification results. All specifications are manually written by the authors and directly injected into the Solidity files. The specifications are written in such a way that they describe the logic of each function as precise as possible. There are 124 reverts_if(), 2 contract invariants, 4 call invariants, 206 function specifications.

The verification results for each project is shown in Table 4.3 under column iContract. The column #V shows the number of specifications that were successfully verified. The column #F shows the number of falsified specifications. The column Time shows the average verification time in seconds. Since we group our specifications into a single specification to compare with Solc-verify, the column #Sp is less than the one shown in Table 4.2. Most of the projects are verified within 5 seconds. Among 336 specification statements, 3 of them are falsified. After manually investigating them, we confirm that iContract exposes contract invariant violations in HOT, QNT and BNB. First, BNB stores the frozen tokens in a mapping called $freezeOf$. When tokens are frozen, they are not subtracted from $totalSupply$. As a result, sum($balances$) $\neq$ $totalSupply$. Second, the $totalSupply$ of QNT remains unchanged even when refresh QNT is created by calling the function $mint$. Again, sum(balances) $\neq$ $totalSupply$. Third, as shown in Figure 4.5, HOT has the following $require$ statement at line 4 which is meant to prevent overflow according to the documentation. However, it also prevents non-overflow cases such as when $\_amount = 0$.

RQ2: To answer RQ2, we compare iContract against Solc-verify, a state-of-the-art tool for verifying function-level properties of smart contracts [32]. Solc-verify is selected as it shares much similarity with iContract, i.e., it supports contract, function and loop invariants. Other verifiers either do not support

```
/// reverts_if(_amount == 0);
function mint(address _to, uint256 _amount) private {
  // Guard against overflow
  require(balances[_to] + _amount > balances[_to]);
  balances[_to] = balances[_to].add(_amount);
}
```

FIGURE 4.5: An example illustrating the effectiveness of reverts_if() in identifying incorrect require statements

user-defined specification (such as VeriSmart [66]), or restrict their specification in specific forms (e.g., linear temporal logic such as in VerX [30] and Smart-Pulse [31]), which are not expressive enough to capture the specification required to verify the correctness of the contracts used in our experiments. We first translate all specifications written in our language to the ones supported by Solc-verify. The translation is not straightforward due to the fact that Solc-verify does not support reverts_if(p) and call invariants. We thus remove the call invariants, reverts_if(p) and convert our ensures(p, q) statements into Solc-verify's specifications. The results are summarized in Table 4.3 under the column *Solc-verify*. While Solc-verify does verify most of the contracts, results inconsistent with ours are reported for 3 contracts, as shown in column #Consistent. All of them are falsified by Solc but are verified by iContract. Our investigation shows that the reason is the missing specifications for external function calls, i.e., the call invariants. In the ZIL project, the external function call $token.transfer$ $(owner, amount)$ transfers tokens to the $owner$. Solc-verify assumes that all global variables are modified after the call and thus sum(balances) $= totalSupply$ is falsified. In contrast, our call invariants indicate that the variable $balances$ is unchanged and the specification sum(balances) $= totalSupply$ is preserved. In the BAT and BNB projects, well-known external functions call such as $send()$ and $transfer()$ are not properly handled in Solc-verify. We remark that besides supporting specification features such as

TABLE 4.3: Comparison against Solc-verify

Project	# Sp	iContract			Solc-Verify			
		#V	#F	Time (s)	#V	#F	Consistent	Time (s)
BAT	13	13	0	4.93	12	1	×	4.51
BNB	15	14	1	7.20	13	2	×	4.00
HT	2	2	0	1.10	2	0	✓	2.77
HOT	17	17	0	1.41	17	0	✓	4.38
IOTX	23	23	0	2.09	23	0	✓	4.26
QNT	11	10	1	1.33	10	1	✓	4.69
MANA	42	42	0	3.94	42	0	✓	6.63
ZIL	40	40	0	4.61	39	1	×	7.13
NXM	22	22	0	2.24	22	0	✓	6.31
SHIB	23	23	0	1.76	23	0	✓	5.95

reverts_if(p) and call invariants, iContract works on Solidity code directly without converting it to another language for verification. This makes verification of the falsified specification statements straightforward in iContract, i.e., by transforming the respective undefined functions into assertions.

4.6 Related Work and Conclusion

The verification for smart contracts has been the interests of multiple researchers. The systems that are closely related to ours are Solc-verify [32] and MVP [101]. Solc-verify translates Solidity contracts into the Boogie intermediate language, and relies on the Boogie system for verification. It supports contract invariant, loop invariant, and pre-/post conditions. In particular, Solc-verify assertion language targets the safety of low-level properties (e.g., the absence of overflows) and high-level contract invariants (e.g., the sum of user balances equates to the total supply). Similarly, Dill *et al.* recently proposed MVP, a static verifier based on the Boogie verifier, for smart contracts in the Move language [101]. MVP supports both contract invariants and functional invariants via pre/post conditions. It also generates global invariants for runtime checking. MVP enables an alias-free memory model through reference elimination which relies on borrow semantics. MVP was deployed for continuous verification on Move code and Diem blockchain. iContract supports all the features supported by Solc-verify and MVP, and additionally supports features like revert and call invariants that are designed to handle dynamic dispatching on unknown function calls.

There are several other verification systems for smart contracts developed in the last few years, e.g., VeriSmart [66], SmartACE [102], and VerX [30]. VeriSmart [66] focuses on intra-procedural analysis for verifying arithmetic (over- and under-flows) safety. The main contribution of their work is an algorithm that could refine transaction invariants of arbitrary transactions. These invariants boost the precision of such verification. However, VeriSmart lacks inter-procedural reasoning. SmartACE [102] is a framework that can verify user-annotated assertions by running multiple independent analysers. It models smart contract library and transforms the verification problem into off-the-self analysers like constrained Horn clause solving (e.g., SeaHorn) for correctness verification. In contrast, iContract presents a built-in static analyser for a rich specification. Finally, VerX [30] focuses on temporal properties of Ethereum contracts. It reduces the temporal safety verification to reachability verification and applies the state-of-the-art reachability checking technique. While temporal logic based specification is useful for specifying global properties, we believe that our specification language is better for supporting the motto of "specification is law" and has its advantage on compositional verification.

To conclude, in this work, we design a static verification system with a specification language which supports fully compositional verification. Using iContract, we have verified 10 high-profile smart contracts against manually developed detailed specifications, many of which are beyond the capacity of existing verifiers. In the future, we intend to improve the performance of iContract further with optimization techniques.

Part II

Smart Contract Vulnerability Repair

Chapter 5

sGuard: Towards Fixing Vulnerable Smart Contracts Automatically

Smart contracts are distributed, self-enforcing programs executing on top of blockchain networks. They have the potential to revolutionize many industries such as financial institutes and supply chains. However, smart contracts are subject to code-based vulnerabilities, which casts a shadow on its applications. As smart contracts are unpatchable (due to the immutability of blockchain), it is essential that smart contracts are guaranteed to be free of vulnerabilities. Unfortunately, smart contract languages such as Solidity are Turing-complete, which implies that verifying them statically is infeasible. Thus, alternative approaches must be developed to provide the guarantee. In this work, we develop an approach which automatically transforms smart contracts so that they are provably free of 4 common kinds of vulnerabilities. The key idea is to apply run-time verification in an efficient and provably correct manner. Experiment results with 5000 smart contracts show that our approach incurs minor run-time overhead in terms of time (i.e., 14.79%) and gas (i.e., 0.79%).

5.1 Introduction

Blockchain is a public list of records which are linked together. Thanks to the underlying cryptography mechanism, the records in the blockchain can resist against modification. Ethereum is a platform which allows programmers to write distributed, self-enforcing programs (a.k.a smart contracts) executing on top of the blockchain network. Smart contracts, once deployed on the blockchain network, become an unchangeable commitment between the involving parties. Because of that, they have the potential to revolutionize many industries such as financial institutes and supply chains. However, like traditional programs, smart contracts are subject to code-based vulnerabilities, which may cause huge financial loss and hinder its applications. The problem is even worse considering that smart contracts are unpatchable once they are deployed on the network. In other words, it is essential that smart contracts are guaranteed to be free of vulnerabilities before they are deployed.

In recent years, researchers have proposed multiple approaches to ensure smart contracts are vulnerability-free. These approaches can be roughly classified into two groups, i.e., verification and testing. However, existing efforts do not provide the required guarantee. Verification of smart contracts is often infeasible since smart contracts are written in Turing-complete programming languages (such as Solidity which is the most popular smart contract language), whereas it is known that testing (of smart contracts or otherwise) only shows the presence not the absence of vulnerabilities.

In this work, we propose an approach and a tool, called sGuard, which automatically fixes potentially vulnerable smart contracts. sGuard is inspired by program fixing techniques for traditional programs such as C or Java, and yet are designed specifically for smart contracts. First, sGuard is designed to guarantee the correctness of the fixes. Existing program fixing approaches (e.g., GenFrog [40], PAR [41], Sapfix [42]) often suffer from the problem of weak specifications, i.e., a test suite is taken as the correctness specification. A fix driven by such a weak correctness criteria may over-fit the given test suites and does not provide correctness guarantee in all cases. Furthermore, fixes for smart contracts may suffer from not only time overhead but also gas overhead (i.e., extra fees for running the additional code) and sGuard is designed to minimize the run-time overhead in terms of time and gas introduced by the fixes.

Given a smart contract, at the high level, sGuard works in two steps. In the first step, sGuard first collects a finite set of symbolic execution traces of the smart contract and then performs static analysis on the collected traces to identify potential vulnerabilities. As of now, sGuard supports 4 types of common vulnerabilities. Note that our static analysis engine is built from scratch as extending existing static analysis engines for smart contracts (e.g., Securify [24] and Ethainter [103]) for our purpose is infeasible. For instance, their sets of semantic rules are incomplete and sometimes produce conflicting results (i.e. a contract both complies and violates a security rule). In addition, they perform abstract interpretation locally (i.e., context/path-insensitive analysis) and thus suffer from many false positives. A contract fixed based on the analysis results from these tools may introduce unnecessary overhead.

In the second step, sGuard applies a specific fixing pattern for each type of vulnerability on the source code to guarantee that the smart contract is free of those vulnerabilities. Our approach is proved to be sound and complete on termination for the vulnerabilities that sGuard supports.

To summarize, our contribution in this work is as follows.

- We propose an approach to fix 4 types of vulnerabilities in smart contracts automatically.
- We prove that our approach is sound and complete for the considered vulnerabilities.
- We implement our approach as a self-contained tool, which is then evaluated with 5000 smart contracts. The experiment results show that sGuard fixes 1605 smart contracts. Furthermore, the fixes incur minor run-time overhead in terms of time (i.e., 14.79% on average) and gas (i.e., 0.79%).

The remainder of the chapter is organized as follows. In Section 5.2, we provide some background about smart contracts and illustrate how our approach works through examples. The problem is then defined formally in Section 5.3. In Section 5.4, we present the details of our approach. The experiment results are presented in Section 5.5. We discuss related work in Section 5.6 and conclude in Section 5.7.

5.2 Background and Overview

In this section, we introduce relevant background on smart contracts and illustrate how our approach addresses the problem of smart contract vulnerabilities through examples.

5.2.1 Vulnerabilities

Just like traditional programs, smart contracts are subject to code-based vulnerabilities. A variety of vulnerabilities have been identified in real-world smart

```
function transferProxy(address from, address to, uint value, uint fee) public {
  if (balances[from] < fee + value) revert();
  uint nonce = nonces[from];
  if (balances[to] + value < balances[to]) revert();
  if (balances[msg.sender] + fee < balances[msg.sender]) revert();
  balances[to] += value;
  balances[msg.sender] += fee;
  balances[from] -= value + fee;
  nonces[from] = nonce + 1;
}
```

(a) Before

```
function transferProxy(address from, address to, uint value, uint fee) public {
  if (balances[from] < add_uint256(fee, value)) revert();
  uint nonce = nonces[from];
  if (add_uint256(balances[to], value) < balances[to]) revert();
  if (add_uint256(balances[msg.sender], fee) < balances[msg.sender]) revert();
  balances[to] = add_uint256(balances[to], value);
  balances[msg.sender] = add_uint256(balances[msg.sender], fee);
  balances[from] = sub_uint256(balances[from], add_uint256(_value, fee))
  nonces[from] = nonce + 1;
}
```

(b) After

FIGURE 5.1: CVE-2018-10376 patched by sGUARD

contracts, some of which have been exploited by attackers and have caused significant financial losses (e.g., [104], [105]). In the following, we introduce two kinds of vulnerabilities through examples.

Example 5.2.1. One category of vulnerabilities is arithmetic vulnerability, e.g., overflow. For instance, in April 2018, an attacker exploited an integer overflow bug in a smart contract named `SmartMesh` and stole a massive amount of tokens (i.e., digital currency). The same bug affected 9 tradable tokens at that time and was named as `ProxyOverflow`. Figure 5.1(a) shows the (simplified) function `transferProxy` in the `SmartMesh` contract which contains the bug. The function is designed for transferring tokens from one account to another, while paying certain fee to the sender (see lines 6 and 7). The developer was apparently aware of potential overflow and introduced relevant checks at lines 2, 4 and 5. Unfortunately, one subtle bug is missed by the checks. That is, if `fee+value` is 0 (due to overflow) and `balances[from]=0`, the attacker is able to bypass the check at line 2 and subsequently increase the balance of `msg.sender` and `to` (see lines 6 and 7) by an amount more than `balances[from]`. During the attack, this bug was exploited to create tokens out of air. *This example highlights that manually-written checks could be error-prone.*

Example 5.2.2. Reentrancy vulnerability is arguably the most infamous vulnerability for smart contracts. It happens when a smart contract C invokes a function of another contract D and subsequently a call back (e.g., through the $fallback$ function in contract D) to contract C is made while it is in an inconsistent state, e.g., the balance of contract C is not updated. Figure 5.2(a) shows a part of a smart contract named `MasBurn` which contains a cross-function reentrancy vulnerability. `MasBurn` implements a Midas protocol token, i.e., a tradable ERC20 token. It allows token holders to burn their owned tokens by sending tokens to a specific `BURN_ADDRESS`, as shown at line 17. The total amount of burned tokens within one week can not exceed `weeklyLimit` (see line 16), which is a variable that limits the amount of tokens to be burned weekly. However, the problem is that the returned value of the function `getThisWeekBurnAmountLeft` (see line 16) has a data dependency on variable `numOfBurns`, and would be wrongly calculated in the case of a reentrancy call at line 17. That is, if the $fallback$ function of the contract at `BURN_ADDRESS` contains a call back to the function `burn`, the function `getThisWeekBurnAmountLeft` is called with an outdated value of `numOfBurns`. As a result, the amount of burned tokens would exceed what is allowed. Although no Ether is lost (or created from air) in such an attack, the (implicit) specification of `MasBurn` is violated in such a scenario. This example also shows the difficulty in handling reentrancy vulnerability, i.e., whether a reentrancy is a vulnerability may depend on the specification of the contract.

5.2.2 Patching Smart Contracts

In the following, we illustrate how SGUARD patches smart contracts through the two examples mentioned above. The technical details are presented in Section 5.4. We remark that SGUARD identifies vulnerabilities based on bytecode while patches them based on the corresponding source code. This is because analysis based on the bytecode is more precise than analysis based on the source code (as the former is not affected by bugs or optimizations in the Solidity compiler), whereas patching at the source code is transparent to the users.

Example 5.2.3. The result of patching the function shown in Figure 5.1(a) using SGUARD is shown in Figure 5.1(b). Almost all arithmetic operations (in statements or expressions) are replaced with function calls that perform the corresponding operations safely (i.e., with proper checks for arithmetic overflow or underflow). This effectively prevents the vulnerability as the function reverts immediately if `fee+value` overflows at line 2. Note that the addition at line 9 is not patched as the variable `nonces` is not deemed critical itself or is depended on by some critical variables.

One might argue that some of the modifications are not necessary, e.g., the one at line 4. This is true for this smart contract, if the goal is to prevent this particular vulnerability. In general, whether a modification is necessary or not can only be answered when the specification of the smart contract is present. SGUARD does not require the specification from the user as that would limit its applicability in practice. sGuard thus always conservatively assumes all arithmetic overflow that may lead to vulnerability are problematic. Although this patch is not minimal, we guarantee that the patched `transferProxy` is free of arithmetic vulnerability.

Example 5.2.4. The result of applying sGuard to the contract shown in Figure 5.2(a) is shown in Figure 5.2(b). sGuard identifies line 17 as an external call, which is critical as an external call invokes a function of another contract which might be under the control of an attacker. sGuard systematically identifies variables that the external call at line 17 depends on (either through control dependency or data dependency). Afterwards, sGuard patches these variables and operations accordingly. In particular, this external call has control dependency on the if-statement at line 5 and is followed by a storage update (`++numOfBurns` at line 18).

- The subtractions at lines 4, 5, 6, 12 are replaced with calls of function `sub_uint256`, which checks underflow.
- The additions at lines 6, 18 are replaced with calls of function `add_uint256` to avoid overflow.
- Function `burn` is patched to prevent reentrancy. That is, we introduce the modifier `nonReentrant` at line 15. This modifier is derived from OpenZeppelin [106], a library for secure smart contract development.

The resultant smart contract is free of arithmetic vulnerability and reentrancy vulnerability.

```
function getThisWeekBurnedAmount() public view returns(uint) {
  uint thisWeekStartTime = getThisWeekStartTime();
  uint total = 0;
  for (uint i = numOfBurns; i >= 1; i--) {
    if (burnTimestampArr[i - 1] < thisWeekStartTime) break;
    total += burnAmountArr[i - 1];
  }
  return total;
}

function getThisWeekBurnAmountLeft() public view returns(uint) {
  return weeklyLimit - getThisWeekBurnedAmount();
}

function burn(uint amount) external payable {
  require(amount <= getThisWeekBurnAmountLeft());
  require(IERC20(tokenAddress).transferFrom(msg.sender,
     BURN_ADDRESS, amount));
  ++numOfBurns;
}
```

(a) Before

```
function getThisWeekBurnedAmount() public view returns(uint) {
  uint thisWeekStartTime = getThisWeekStartTime();
  uint total = 0;
  for (uint i = numOfBurns; i >= 1; (i = sub_uint256(i, 1))) {
    if (burnTimestampArr[sub_uint256(i, 1)] < thisWeekStartTime)
       break;
    total = add_uint256(total, burnAmountArr[sub_uint256(i, 1)]);
  }
  return total;
}

function getThisWeekBurnAmountLeft() public view returns(uint) {
  return sub_uint256(weeklyLimit, getThisWeekBurnedAmount());
}

function burn(uint amount) external payable nonReentrant {
  require(amount <= getThisWeekBurnAmountLeft());
  require(IERC20(tokenAddress).transferFrom(msg.sender,
     BURN_ADDRESS, amount));
  (numOfBurns = add_uint256(numOfBurns, 1));
}
```

(b) After

FIGURE 5.2: MasBurn patched by sGuard

5.3 Problem Definition

In the following, we first present the semantics for Solidity smart contracts, and then define our problem.

5.3.1 Concrete Semantics

A smart contract $\mathcal{S}$ can be viewed as a finite state machine $\mathcal{S} = (Var, init, N, i, E)$ where Var is a set of variables; $init$ is the initial valuation of the variables; N is a finite set of control locations; $i \in N$ is the initial control location, i.e., the start of the contract; and $E \subseteq N \times C \times N$ is a set of labeled edges, each of which is of the form (n, c, n') where c is an opcode. There are a total of 78 opcodes in Solidity (as of version 0.5.3), as summarized in Table 5.1. Note that each opcode is statically assigned with a unique program counter, i.e., each opcode can be uniquely identified based on the program counter.

Note that Var includes stack variables, memory variables, and storage variables. Stack variables are mostly used to store primitive values and memory variables are used to store array-like values (declared explicitly with keyword $memory$). Both stack and memory variables are volatile, i.e., they are cleared after each transaction. In contrast, storage variables are non-volatile, i.e., they are persistent on the blockchain. Together, the variables' values identify the state of the smart contract at a specific point of time. At the Solidity source code level, stack and memory variables can be considered as local variables in a specific function; and storage variables can be considered as contract-level variables.

A concrete trace of the smart contract is an alternating sequence of states and opcodes $\langle s_0, op_0, s_1, op_1, \cdots \rangle$ such that each state s_i is of the form (pc_i, S_i, M_i, R_i) where $pc_i \in N$ is the program counter; S_i is the valuation of the stack variables;

M_i is the valuation of the memory variables; and R_i is the valuation of the storage variables. Note that the initial state s_0 is $(0, S_0, M_0, R_0)$ where S_0, M_0 and R_0 are the initial valuation of the variables defined by $init$. Furthermore, for all i, $(pc_{i+1}, S_{i+1}, M_{i+1}, R_{i+1})$ is the result of executing opcode op_i given the state (pc_i, S_i, M_i, R_i) according to the semantic of op_i. The semantics of opcodes are shown in Figure 5.3 in the form of execution rules, each of which is associated with a specific opcode. Each rule is composed of multiple conditions above the line and a state change below the line. The state change is read from left to right, i.e., the state on the left changes to the state on the right if the conditions above the line are satisfied. Note that this formal semantics is based on the recent effort on formalizing Etherum [107].

Most of the rules are self-explanatory and thus we skip the details and refer the readers to [107]. It is worth mentioning how external calls are abstracted in our semantic model. Given an external function call (i.e., opcode CALL), the execution temporarily switches to an execution of the invoked contract. The result of the external call, abstracted as res, is pushed to the stack.

5.3.2 Symbolic Semantics

In order to define our problem, we must define the kinds of vulnerabilities that we focus on. Intuitively, we say that a smart contract suffers from certain vulnerability if there exists an execution of the smart contract that satisfies certain constraints. In the following, we extend the concrete traces to define symbolic traces of a smart contract so that we can define whether a symbolic trace suffers from certain vulnerability.

To define symbolic traces, we first extend the concrete values to symbolic values. Formally, a symbolic value has the form of $op(operand_0, \cdots, operand_n)$

$$\frac{}{(pc, S, M, R) \rightsquigarrow \Box}\ \text{STOP} \qquad \frac{S_1, x = S.pop()}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_1, M, R)}\ \text{POP}$$

$$\frac{S_1, x = S.pop()\ z = op(x)\ S_2 = S_1.push(z)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R)}\ \text{UNARY-OP}$$

$$\frac{\begin{array}{c} S_1, x = S.pop()\ S_2, y = S_1.pop() \\ z = op(x, y)\ S_3 = S_2.push(z) \end{array}}{(pc, S, M, R, pc) \rightsquigarrow (pc + 1, S_3, M, R, pc + 1)}\ \text{BINARY-OP}$$

$$\frac{\begin{array}{c} S_1, x = S.pop()\ S_2, y = S_1.pop()\ S_3, m = S_2.pop() \\ z = op(x, y, m)\ S_4 = S_3.push(z) \end{array}}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_4, M, R)}\ \text{TERNARY-OP}$$

$$\frac{S_1, p = S.pop()\ v = M[p]\ S_2 = S_1.push(v)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R)}\ \text{MLOAD}$$

$$\frac{S_1, p = S.pop()\ S_2, v = S_1.pop()\ M_1 = M[p \leftarrow v]}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M_1, R)}\ \text{MSTORE}$$

$$\frac{S_1, p = S.pop()\ v = R[p]\ S_2 = S_1.push(v)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R)}\ \text{SLOAD}$$

$$\frac{S_1, p = S.pop()\ S_2, v = S_1.pop()\ R_1 = R[p \leftarrow v]}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R_1)}\ \text{SSTORE}$$

$$\frac{v = S.get(i)\ S_1 = S.push(v)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_1, M, R)}\ \text{DUP-I}$$

$$\frac{v_0 = S.get(0)\ v_i = S.get(i)\ S_1 = S[0 \leftarrow v_i]\ S_2 = S_1[i \leftarrow v_0]}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R)}\ \text{SWAP-I}$$

$$\frac{S_1, lbl = S.pop()\ S_2, c = S_1.pop()\ c \neq 0}{(pc, S, M, R) \rightsquigarrow (lbl, S_2, M, R)}\ \text{JUMPI-T}$$

$$\frac{S_1, lbl = S.pop()\ S_2, c = S_1.pop()\ c = 0}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_2, M, R)}\ \text{JUMPI-F}$$

$$\frac{S_1, lbl = S.pop()}{(pc, S, M, R) \rightsquigarrow (lbl, S_1, M, R)}\ \text{JUMP} \qquad \frac{res = call()\ S_1 = S.push(res)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_1, M, R)}\ \text{CALL}$$

$$\frac{S_1, p = S.pop()\ S_2, n = S_1.pop()\ v = sha3(M[p, p + n])\ S_3 = S_2.push(v)}{(pc, S, M, R) \rightsquigarrow (pc + 1, S_3, M, R)}\ \text{SHA3}$$

FIGURE 5.3: Operational semantics of Ethereum opcodes. *pop*, *push*, and *get* are self-explanatory stack operations. $m[p \leftarrow v]$ denote an operations which returns the same stack/mapping as m except that the value of position/key p is changed to v. Rule UNARY-OP (BINARY-OP, TERNARY-OP) applies to all unary (binary, ternary) operations; rule DUP-I, applies to all duplicate operations; and rule SWAP-I applies to all swap operations.

Rule	Opcodes
STOP	`SELFDESTRUCT, REVERT, INVALID, RETURN, STOP`
POP	`POP`
UNARY-op	`NOT, ISZERO, CALLDATALOAD, EXTCODESIZE, BLOCKHASH, BALANCE, EXTCODEHASH`
BINARY-op	`ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, EXP, SIGNEXTEND, LT, GT, SLT, SGT, EQ, AND, OR, XOR, BYTE, SHL, SHR, SAR`
TERNARY-op	`ADDMOD, MULMOD, CALLDATACOPY, CODECOPY, RETURNDATACOPY`
MLOAD	`MLOAD`
SHA3	`SHA3`
MSTORE	`MSTORE, MSTORE8`
SLOAD	`SLOAD`
SSTORE	`SSTORE`
DUP-I	`DUP1`$\cdots$`DUP16`
SWAP-I	`SWAP1`$\cdots$`SWAP16`
JUMPI-T/JUMPI-F	`JUMPI`
JUMP	`JUMP`
CALL	`STATICCALL, CALL, CALLCODE, CREATE, CREATE2, DELEGATECALL, SELFDESTRUCT`

TABLE 5.1: The opcodes according to each rule

where *op* is an opcode and $operand_0, \cdots, operand_n$ are the operands. Each operand may be a concrete value (e.g., an integer number or an address) or a symbolic value. Note that if all operands of an opcode are concrete values, the symbolic value is a concrete value as well, i.e., the result of applying *op* to the concrete operands. For instance, `ADD(5, 6)` is 11. Otherwise, the value is symbolic. One exception is that if *op* is `MLOAD` or `SLOAD`, the result is symbolic even if the operands are concrete, as it is not trivial to maintain the concrete content of the memory or storage. For instance, loading a value from address `0x00` from the storage results in the symbolic value `SLOAD(0x00)` and increasing the value at storage address `0x00` by 6 results in a symbolic value `ADD(SLOAD(0x00),0x06)`. For another instance, the result of symbolically executing `SHA3(n,p)` is `SHA3(MLOAD(n,p))`, i.e., the SHA3 hash of the value located from address n to $n + p$ in the memory.

With the above, a symbolic trace is an alternating sequence of states and

```
function transfer(address _to, uint _value) public {
  if (_value <= 0) revert();
  if (balances[msg.sender] < _value) revert();
  if (balances[_to] + _value < balances[_to]) revert();
  balances[msg.sender] = balances[msg.sender] - _value;
  balances[_to] = balances[_to] + _value;
}
```

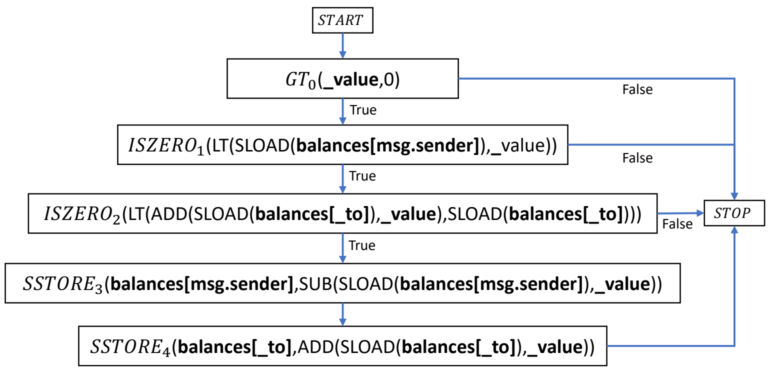

START
GT0(_value,0)
True
False
ISZERO1(LT(SLOAD(balances[msg.sender]),_value))
True
False
ISZERO2(LT(ADD(SLOAD(balances[_to]),_value),SLOAD(balances[_to])))
False
STOP
True
SSTORE3(balances[msg.sender],SUB(SLOAD(balances[msg.sender]),_value))
SSTORE4(balances[_to],ADD(SLOAD(balances[_to]),_value))

FIGURE 5.4: An example of control and data dependency

opcodes $\langle s_0, op_0, s_1, op_1, \cdots \rangle$ such that each state s_i is of the form $(pc_i, S_i^s, M_i^s, R_i^s)$ where pc_i is the program counter; S_i^s, M_i^s and R_i^s are the valuations of stack, memory and storage respectively. Note that S_i^s, M_i^s and R_i^s may hold symbolic values as well as concrete ones. For all i, $(pc_{i+1}, S_{i+1}^s, M_{i+1}^s, R_{i+1}^s)$ is the result of executing opcode op_i symbolically given the state $(pc_i, S_i^s, M_i^s, R_i^s)$.

A symbolic execution engine is one which systematically generate the symbolic traces of a smart contract. Note that different from concrete execution, a symbolic execution would generate two traces given an `if`-statement, one visits the then-branch and the other visits the else-branch. Furthermore, in the case of an external call (i.e., `CALL`), instead of switching the current execution context to another smart contract, we can simply use a symbolic value to represent the returned value of the external call.

5.3.3 Problem Definition

Intuitively, a vulnerability occurs when there are dependencies from certain critical instructions (e.g., `CALL` and `DELEGATECALL`) to a set of specific instructions (e.g., `ADD`, `SUB` and `SSTORE`). Therefore, to define our problem, we first define (control and data) dependency, based on which we define the vulnerabilities.

Definition 1 (Control dependency)**.** An opcode op_j is said to be control-dependent on op_i if there exists an execution from op_i to op_j such that op_j post-dominates all op_k in the path from op_i to op_j (excluding op_i) but does not post-dominates op_i. An opcode op_j is said to post-dominate an opcode op_i if all traces starting from op_i must go through op_j.

Figure 5.4 illustrates an example of control dependency. The source code is shown on the top and the corresponding control flow graph is shown on the bottom. All variables and their symbolic values are summarized in Table 5.2. The source code presents secure steps to transfer `_value` tokens from `msg.sender` account to `_to` account. There are 3 then-branches followed by 2 storage updates. According to the definition, both $\texttt{SSTORE}_3$ and $\texttt{SSTORE}_4$ are control-dependent on $\texttt{ISZERO}_1$, $\texttt{ISZERO}_2$ and $\texttt{GT}_0$.

Definition 2 (Data dependency)**.** An opcode op_j is said to be data-dependent on op_i if there exists a trace which executes op_i and subsequently op_j such that $W(op_i) \cap R(op_j) \neq \emptyset$ where $R(op_j)$ is a set of locations read by op_j; $W(op_i)$ is a set of locations written by op_i.

Figure 5.4 also illustrates an example of data dependency. Opcode $\texttt{ISZERO}_1$ and $\texttt{ISZERO}_2$ are data-dependent on $\texttt{SSTORE}_3$ and $\texttt{SSTORE}_4$. It has 2 traces, i.e.,

Variable	Symbolic Value
$\_to$	`CALLDATALOAD(0x04)`
$\_value$	`CALLDATALOAD(0x24)`
$balances[msg.sender]$	`SHA3(MLOAD(0x00,0x40))`
$balances[\_to]$	`SHA3(MLOAD(0x00,0x40))`

TABLE 5.2: Variables and their symbolic values of Figure 5.4

one trace loads data from storage address SHA3(MLOAD(0x00,0x40)) which is written by SSTORE$_1$ and SSTORE$_2$ in another trace.

We say an opcode op_j is dependent on opcode op_i if op_j is control or data dependent on op_i or op_j is dependent on an opcode op_k such that op_k is dependent on op_i.

Vulnerabilities: In the following, we define the 4 kinds of vulnerabilities that we focus on, i.e., intra-function and cross-function reentrancy, dangerous tx.origin and arithmetic overflow. We remark that while we can certainly detect more kinds of vulnerabilities, it is not always clear how to fix them, i.e., it may not be feasible to know the intended behavior. For example, in the case of fixing an *accessible selfdestruct* vulnerability (i.e., a smart contract suffers from this vulnerability if it may be destructed by anyone [103]), we would not know for sure who should have the privilege to access selfdestruct.

Let C be a set of critical opcodes which contains CALL, CALLCODE, DELEGATECALL, SELFDESTRUCT, CREATE and CREATE2, i.e., the set of all opcode associated with external calls except STATICCALL. The reason that STATICCALL is excluded from C is that STATICCALL can not update storage variables of the called smart contract and thus is considered to be safe.

Definition 3 (Intra-function reentrancy vulnerability)**.** A symbolic trace suffers from intra-function reentrancy vulnerability if it executes an opcode $op_c \in C$

```
uint numWithdraw = 0;
function withdraw() external {
    uint256 amount = balances[msg.sender];
    balances[msg.sender] = 0;
    (bool ret, ) = msg.sender.call.value(amount)("");
    require(ret);
    numWithdraw ++;
}
```

FIGURE 5.5: A non-reentrant case captured by NW

and subsequently executes an opcode op_s in the same function such that op_s is `SSTORE`, and op_c depends on op_s.

A smart contract suffers from intra-function reentrancy vulnerability if and only if at least one of its symbolic traces suffers from intra-function reentrancy vulnerability. The above definition is inspired from the *no writes after call* (NW) property [24]. It is however more accurate than NW, as it avoids violations of NW which are not considered as reentrancy vulnerability. For instance, the function shown in Figure 5.5 violates NW, although it is not subject to reentrancy vulnerability. It is because the external call `msg.sender.call` has no dependency on `numWithdraw`. In other words, there does not exist a dependency from op_c to op_s.

Definition 4 (Cross-function reentrancy vulnerability)**.** A symbolic trace tr suffers from cross-function reentrancy vulnerability if it executes an opcode op_s where op_s is `SSTORE` and there exists a symbolic trace tr' subject to intra-function reentrancy vulnerability such that the opcode op_c of tr' depends on op_s, and they belong to different functions.

A smart contract suffers from cross-function reentrancy vulnerability if and only if at least one of its symbolic traces suffers from cross-function reentrancy vulnerability. This vulnerability differs from intra-function reentrancy as the

```
1 function transfer(address to, uint amount) external {
2     if (balances[msg.sender] >= amount) {
3       balances[to] += amount;
4       balances[msg.sender] -= amount;
5     }
6 }
7
8 function withdraw() external nonReentrant {
9     uint256 amount = balances[msg.sender];
10    (bool ret, ) = msg.sender.call.value(amount)("");
11    require(ret);
12    balances[msg.sender] = 0;
13 }
```

FIGURE 5.6: An example of cross-function reentrancy vulnerability

```
function sendTo(address receiver, uint amount) public {
    require(tx.origin == owner);
    receiver.transfer(amount);
}
```

FIGURE 5.7: An example of dangerous tx.origin vulnerability

attacker launches an attack through two different functions, which makes it harder to detect. Figure 5.6 shows an example of cross-function reentrancy. The developer is apparently aware of intra-function reentrancy and thus add the modifier `nonReentrant` to the function `withdraw` for preventing reentrancy. However, reentrancy is still possible through function `transfer`, in which case the attacker is able to double his Ether. That is, the attacker receives Ether at line 10 and illegally transfers it to another account at line 3. Although cross-function reentrancy vulnerabilities were described in Sereum [64] and Consensys [108], our work is the first work to define it formally.

Definition 5 (Dangerous tx.origin vulnerability). A symbolic trace suffers from dangerous tx.origin vulnerability if it executes an opcode $op_c \in C$ which depends on an opcode `ORIGIN`.

A smart contract suffers from dangerous `tx.origin` vulnerability if and only if at least one of its symbolic traces suffer from dangerous tx.origin vulnerability. This vulnerability happens due to an incorrect usage of the global variable `tx.origin` to authorize a user. An attack happens when a user U sends a transaction to a malicious contract A, which intentionally forwards this transaction to a contract B that relies on a vulnerable authorization check (e.g., `require(tx.origin == owner)`). Since `tx.origin` returns the address of U, contract A successfully impersonates U. Figure 5.7 presents an example suffering from dangerous tx.orgin vulnerability, i.e., a malicious contract may impersonate the owner to withdraw all Ethers.

Definition 6 (Arithmetic vulnerability)**.** A symbolic trace suffers from arithmetic vulnerability if it executes an opcode op_c in C and op_c depends on an opcode op_a which is `ADD`, `SUB`, `MUL` or `DIV`.

A smart contract suffers from arithmetic vulnerability if and only if at least one of its symbolic traces suffer from arithmetic vulnerability. Intuitively, this vulnerability occurs when an external call data-depends on an arithmetic operation (e.g., addition, subtraction, or multiplication). For instance, the example in the Figure 5.2 is vulnerable due to the presence of data dependency between the external call at line 17 and the expression `weeklyLimit - getThisWeekBurnedAmount()` at line 12. Arithmetic vulnerabilities are the target of multiple tools designed for vulnerability detection. In general, arithmetic vulnerability detection using static analysis often results in high false positive. Therefore, tools such as Securify [24] and Ethainter [103] do not support this vulnerability in spite of its importance. In the above definition, we focus on only critical arithmetic operations to reduce false positives. That is, an arithmetic operation is not considered critical as long as the smart contract does not spread its wrong computation to other smart contracts through external calls. For instance, wrong

ERC20 token transfer (e.g., CVE-2018-10376) is not critical because it can be reverted by the contract's admin, whereas wrong Ether transfer is irreversible.

Problem definition: Our problem is then defined as follows. Given a smart contract S, construct a smart contract T such that T satisfies the following.

- Soundness: T is free of any of the above vulnerabilities.
- Preciseness: For every symbolic trace tr of S, if tr does not suffer from any of the vulnerabilities, there exists a symbolic trace tr' in T which, given the same inputs, produces the same outputs and states.
- Efficiency: T's execution speed and gas consumption are minimally different from those of S.

Note that the first two are about the correctness of construction, whereas the last one is about the performance in terms of computation and gas overhead.

5.4 Detailed Approach

In this section, we present the details of our approach. The key challenge is to precisely identify where vulnerabilities might arise and fix them accordingly. Note that precisely identifying control/data-dependency is a prerequisite for precisely identifying vulnerabilities. One approach to identify vulnerabilities is through static analysis based on over-approximation. For instance, multiple existing tools (e.g., Securify [24] and Ethainter [103]) over-approximate Etherum semantics using rewriting rules and leverage rewriting systems such as Datalog to identify vulnerabilities through critical pattern matching. While useful (and typically efficient) in detecting vulnerabilities, such approaches are not ideal

Algorithm 3: $sGuard$

```
1 establish a bound for each loop;
2 enumerate symbolic traces $Tr$;
3 foreach trace $tr$ in $Tr$ do
4     let $dp \leftarrow dependency(tr)$;
5     $fixReentrancy(tr, dp)$;
6     $fixTxOriginAndArithemic(tr, dp)$;
```

for our purpose for multiple reasons. First, there are often many false alarms as they perform abstract interpretation locally (i.e., context/path-insensitive analysis). In our setting, once a vulnerability is identified, we fix it by introducing additional run-time checks. False alarms thus translate to runtime overhead in terms of both time and gas. Second, existing approaches are often incomplete, i.e., not all dependencies are captured. For instance, Securify ignores data dependency through storage variables, i.e., the dependency due to `SSTORE(c,b)` is lost if c is not a constant, whereas Ethainter ignores control dependency completely. Thirdly, rewriting systems such as Datalog may terminate without any result, in which case the analysis result may not be sound. Therefore, in our work, we propose an algorithm which covers all dependencies with high precision and always terminates with the correct result.

The details of our algorithm are shown in Algorithm 3. From a high-level point of view, it works as follows. First, symbolic traces are systematically enumerated, up to certain threshold number of iterations for each loop. Second, each symbolic trace is checked to see whether it is subject to certain vulnerability according to our definitions. Lastly, the corresponding source code of the vulnerability is identified based on the AST and fixed. In the following, we present details of each step one-by-one.

5.4.1 Enumerating Symbolic Traces

Note that our definitions of vulnerabilities are based on symbolic traces. Thus, in this first step, we set out to collect a set of symbolic traces Tr. As defined in Section 5.3.2, a symbolic trace is a sequence of the form $\langle s_0, op_0, \cdots, s_n, op_n, s_{n+1}\rangle$. In the following, we focus on symbolic traces that are maximum, i.e., the last opcode op_n is either `REVERT`, `INVALID`, `SELFDESTRUCT`, `RETURN`, or `STOP`.

Systematically generating the maximum symbolic traces is straightforward in the absence of loops, i.e., we simply apply the symbolic semantic rules iteratively until it terminates. In the presence of loops, however, as the condition to exit the loop is often symbolic, this procedure would not terminate. This is a well-known problem for symbolic execution and the remedy is typically to bound the number of iterations heuristically. Such an approach however does not work in our setting, since we must identify all data/control dependency to identify all potential vulnerabilities. In the following, we establish a bound on the number of iterations on the loops which we prove is sufficient for identifying the vulnerabilities that we focus on.

Given a smart contract $\mathcal{S} = (Var, init, N, i, E)$, a loop is in general a strongly connected component in $\mathcal{S}$. Thanks to structural programming, we can always identify the loop heads, i.e., the control location where a *while*-loop starts or a recursive function is invoked. In the following, we associate each location $n \in N$ with a bound, denoted as $bound(n)$. If n is a loop head, $bound(n)$ intuitively means how many times n has to be visited in at least one of symbolic traces we collect. If n is not part of any strongly connected component, we have $bound(n) = 1$. Otherwise, $bound(n)$ is defined as follows.

- If $(n, op_n, n') \in E$ and n' is the loop head, $bound(n) = 0$ if op_n is not an assignment; otherwise $bound(n) = 1$.

- If $(n, op_n, n') \in E$, n' is not the loop head and there is no m such that $(n, op_n, m) \in E$, i.e., n is not branching, $bound(n) = bound(n')$ if op_n is not an assignment; otherwise $bound(n) = bound(n') + 1$.
- If $(n, op_n, m_0) \in E$ and $(n, op_n, m_1) \in E$, i.e., n is branching, $bound(n) = bound(m_1) + bound(m_2)$.

Intuitively, the bound of a loop head is computed based on the number of branching statements and assignment statements inside the loop. That is, the bound of a loop head n can be computed by traversing the CFG in the reverse order, i.e., from the exiting nodes of the loop to n. Every execution path maintains a bound, which equals to the number of assignment statements in that path. If two execution paths meet at a branching statement then the new bound is set to the sum of their bounds. In our implementation, the bounds for every node $n \in N$ are statically computed using a fixed-point algorithm, with a complexity of $\mathcal{O}((\#N)^2)$ where $\#N$ is the number of nodes. Once the bounds are computed, we systematically enumerate all maximum symbolic traces such that each loop head n is visited at most $bound(n)$ times. It is straightforward to see that this procedure always terminates and returns a finite set of symbolic traces.

Example 5.4.1. In the following, we illustrate how $bound(x < 100)$ is computed. The example is shown in the Figure 5.8 where the graph on the right represents the source code on the left (a.k.a. control flow graph which can be constructed using existing approaches [43]). Assignment statements are highlighted in blue. There is a total of 3 paths $P1, P2, P3$ in the *while-loop*, and they visit 5 assignment statements. Since we follow both branches of an if-statement, there exists a symbolic trace tr containing $P1, P2, P3$ regardless of the order. Trace tr is of the form $\langle \cdots, op_i, \cdots, op_j, \cdots, op_k, \cdots, op'_i, \cdots \rangle$ where op_i and op'_i are executed

```
function transfer(
  uint x, uint y,
  uint z, uint m, uint n
) {
  while (x < 100) {
    x = y + 1;
    if (y < 100) {
      y = z + 1;
      if (z < 100) {
        z = m + 1;
      } else {
        m = n + 1;
      }
    } else {
      n = x + 1;
    }
  }
  msg.sender.send(x);
}
```

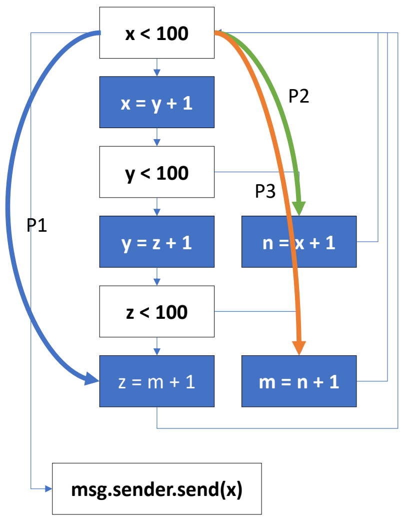


FIGURE 5.8: An example on how the $bound(n)$ is computed

opcodes of the loop head $x < 100$; op_j is mapped to $y < 100$ and op_k is mapped to $z < 100$.

There are 5 assignment statements between op_i and op'_i and the bound of the loop head is 5. Note that the number of assignment statements in the example is the number of `SWAP`s appeared in between op_i and op'_i.

The following establishes the soundness of our approach, i.e., using the bounds, we are guaranteed to never miss any of the 4 kinds of vulnerabilities that we focus on.

Lemma 1. Given a smart contract, if there exists a symbolic trace which suffers from intra-function reentrancy vulnerability (or cross-function reentrancy, or dangerous tx.origin, or arithmetic vulnerability), there must be one in Tr. □

We sketch the proof in the following. All vulnerabilities in Section 5.3 are defined based on control/data dependency between opcodes. That means we always have a vulnerable trace, if there is, one as long as the set of symbolic traces we collect exhibit all possible dependency between opcodes. To see that all dependencies are exhibited in the traces we collect, we distinguish two cases. All control dependency between opcodes are identified as long as all possible branches in the smart contract are executed. This condition is satisfied based on the way we collect traces in Tr. This argument applies to data dependency between opcodes which do not belong to any loop as well. Next, we consider the data dependency between opcodes inside a loop. Note that with each loop iteration, there are two possible cases: no new data dependency is identified (i.e., the data dependency reaches fixed point) or at least 1 new dependency is identified. If the loop contains n assignments, in the worst case, all of these opcodes depend on each other and we need a trace with n iterations to identify all of them. Based on how we compute the bound for the loop heads, the trace is guaranteed to be in Tr. Thus, we establish that the above lemma is proved.

It is well-known that symbolic execution engines may suffer from the path explosion problem. sGuard is not immune as well, i.e., the number of symbolic paths explored by sGuard is in general exponential in the loop bounds. Existing symbolic execution engines address the problem by allowing users to configure a bound K which is the maximum number of times any loop is unrolled. In practice, it is highly non-trivial to know what K value should be used. Given the impact of K, i.e., the number of paths are exponential in the value of K, existing tools often set K to be a small number by default, such as 3 in sCompile [109] and 5 in Manticore [110]; and it is unlikely that users would configure it differently. While having a large K leads to the path explosion problem, having a small K leads to false negatives. For instance, with $K = 3$, the overflow vulnerabilities due to the two expressions $m = n + 1$, $n = x + 1$ in the Figure 5.8 would be missed as this bound is not sufficient to infer dependency from variable x on m and n. In contrast, sGuard automatically identifies a loop bound for

Algorithm 4: build CFG

```
1 let $edges \leftarrow \emptyset$;
2 foreach trace tr in Tr do
3     foreach $op_i, pc_i$ in tr do
4         if $op_i$ = $JUMPI$ then
5             let $edge \leftarrow (pc_i, pc_{i+1})$ ;
6             add $edge$ to $edges$;
7 return $edges$;
```

each loop which guarantees that no vulnerabilities are missed. In Section 5.5, we empirically evaluate whether the path explosion problem occurs often in practice.

5.4.2 Dependency Analysis

Given the set of symbolic traces Tr, we then identify dependency between all opcodes in every symbolic trace in Tr, with the aim to check whether the trace suffers from any vulnerability. In the following, we present our approach to capture dependency from symbolic traces.

Given a symbolic trace Tr, an opcode op_i, we aim to identify a set of opcodes dp in Tr such that: (soundness) for all op_k in dp, op_i depends on op_k; and (completeness) for all op_k in Tr, if op_i depends on op_k then $op_k \in dp$. To identify dp, we systematically identify all opcodes that op_i is control-dependent on in Tr, all opcodes that op_i is data-dependent on in Tr and then compute their transitive closure.

To systematically identify all control-dependency, we build a control flow graph (CFG) from Tr (as shown in Algorithm 4). Afterwards, we build a post-dominator tree based on the CFG using a workList algorithm [111]. The result is a set $PD(op_i)$ which are the opcodes that post-dominate op_i. The set of opcodes which op_i control-depend on in the symbolic trace tr is then systematically identified as the following.

$$\{ \ op \mid op \in tr; \exists \ (op_m, op_n) \in succs(op), op_i \in PD(op_m), op_i \notin PD(op_n) \ \}$$

where $succs(op)$ returns successors of op according to CFG.

Identifying the set of opcodes which op_i is data-dependent on is more complicated. Data dependency arises from 3 data sources, i.e., stack, memory and storage. In the following, we present our over-approximation based algorithm which traces data-flow on these data sources in order to capture data dependency. Although an opcode typically reads and writes data to the same data source, an opcode may write data to a different data source in some cases. That makes data-flow tracing complicated, i.e., data flows from stack to memory through `MSTORE`, memory to stack through `MLOAD`, stack to storage through `SSTORE` and storage to stack through `SLOAD`. Since only assignment opcodes (i.e., `SWAP`, `MSTORE`, and `SSTORE`) create data dependency, we thus design an algorithm to identify data-dependency based on the assignment opcodes in tr. The details are presented in the Algorithm 5, which takes a symbolic trace tr and opcode op_i as input and returns a set of opcodes that op_i is data-dependent on.

Algorithm 5 systematically identifies those opcodes in tr which taint op_i. An opcode op_j is said to taint another opcode op_i if op_i reads data from stack indexes written by op_j, or there exists an opcode op_t such that op_j taints op_t and op_t taints op_i. For each op_j that taints op_i, there are three possible dependency cases.

Algorithm 5: $f_d(tr, op_i)$

```
1  let $opcodes \leftarrow \emptyset$;
2  foreach $op_j$ that taints $op_i$ do
3      if $op_j$ is an assignment opcode then
4          add $op_j$ to $opcodes$ ;
5      if $op_j$ reads data from memory which was written by an assignment opcode
         $op_k$ then
6          add $op_k$ to $opcodes$ ;
7          add $f_d(tr, op_k)$ to $opcodes$;
8      if $op_j$ reads data from storage which was written by an assignment opcode
         $op_k$ then
9          if $op_k$ is not visited then
10             add $op_k$ to $opcodes$;
11             foreach trace $tr'$ contains $op_k$ do
12                 add $f_d(tr', op_k)$ to $opcodes$;
13 return $opcodes$;
```

- Stack dependency: op_i is data-dependent on op_j if op_j is an assignment opcode (i.e., `SWAP`) (lines 3-4)

- Memory dependency: op_j is data-dependent on op_k if op_j reads data from memory which was written by the assignment opcode op_k (i.e., `MSTORE`) (lines 5-7)

- Storage dependency: op_j is data-dependent on op_k if op_j reads data from storage which was written by the assignment opcode op_k (i.e., `SSTORE`) (lines 8-12)

Note that the algorithm is recursive, i.e., if op_k is added into the set of opcodes to be returned, a recursive call is made to further identify those opcode that op_k is data-dependent on (lines 7 and 12). Further note that since storage is globally accessible, the analysis may be cross different traces in Tr (line 11).

Algorithm 5 in general over-approximates. For instance, because memory and storage addresses are likely symbolic values, a reading address and a writing address are often incomparable, in which case we conservatively assume that the addresses may be the same. In other words, $R(op_j) \cap W(op_k) \neq \emptyset$ is true if either $R(op_j)$ or $W(op_k)$ is a symbolic address.

5.4.3 Fixing the Smart Contract

Once the dependencies are identified, we check whether each symbolic tr suffers from any of the vulnerabilities defined in Section 5.3.3 and then fix the smart contract accordingly. In general, a smart contract is fixed as follows. Given a vulnerable trace tr, according to our definitions in Section 5.3.3, there must be an external call $op_c \in C$ in tr. Furthermore, there must be some other opcode *op* that op_c depends on which together makes tr vulnerable (e.g., if *op* is `SSTORE`, tr suffers from reentrancy vulnerability; if *op* is `ADD`, `SUB`, `MUL` or `DIV`, tr suffers from arithmetic vulnerability). The idea is to introduce runtime checks right before *op* so as to prevent the vulnerability. According to the type of vulnerability, the runtime checks are injected as follows.

- To prevent intra-function reentrancy vulnerability, we add a modifier `nonReentrant` to the function F containing *op*. Note that the `nonReentrant` modifier works as a mutex which blocks an attacker from re-entering F. To prevent cross-function reentrancy vulnerability, we add the modifier `nonReentrant` to the function containing *op*. The details of the fixing algorithm are presented in Algorithm 6 which takes a vulnerable trace tr and the dependency relation dp as inputs.

- To fix dangerous tx.origin vulnerability, we replace *op* (i.e., `ORIGIN`) with `msg.sender` which returns address of the immediate account that invokes the function.
- To fix arithmetic vulnerability, we replace *op* (i.e., `ADD`, `SUB`, `MUL`, `DIV`, or `EXP`) with a call to a safe math function which checks for overflow/underflow before performing the arithmetic operation.

Note that in the case of reentrancy vulnerability and arithmetic vulnerability, if a runtime check fails (e.g., `assert(x > y)` which is introduced before `x - y` fails), the transaction reverts immediately and thus the vulnerability is prevent, although the gas spent on executing the transaction so far would be wasted. Further note while Algorithm 6 is applied to every vulnerable trace, the same fix (e.g., introducing `nonReentrant` on the same function) is applied once. We refer the readers to Section 5.2.2 for examples on how smart contracts are fixed.

Algorithm 6: $fixReentrancy(tr, dp)$

```
1 let $tr \leftarrow \langle s_0, op_0, \cdots, s_n, op_n, s_{n+1} \rangle$;
2 foreach $i$ in $0..n$ do
3     if $op_i \in C$ then
4         foreach $j$ in $i+1..n$ do
5             if $op_j$ is $SSTORE$ and $op_i$ depends on $op_j$ according to $dp$ then
                  /* Fix intra-function reentrancy */
6                 add modifier nonReentrant to the function containing $op_i$;
                  /* Fix cross-function reentrancy */
7                 foreach $op_s$ that $op_i$ depends on according to $dp$ do
8                     if $op_s$ is $SSTORE$ then
9                         add modifier nonReentrant to the function containing $op_s$;
```

The following establishes the soundness of our approach.

Theorem 1. A smart contract fixed by Algorithm 3 is free of intra-function reentrancy vulnerability, cross-function reentrancy vulnerability, dangerous tx.origin vulnerability, and arithmetic vulnerability. □

The proof of the theorem is sketched as follows. According to the Lemma 1, given a smart contract S, if there are vulnerable traces, at least one of them is identified by sGuard. Given how sGuard fixes each kind of vulnerability, fixing all vulnerable traces in Tr implies that all vulnerable traces are fixed in S.

We acknowledge that our approach does not achieve the preciseness as discussed in Section 5.3.3. That is, a trace which is not vulnerable may be affected by the fixes if it shares some opcodes with the vulnerable traces. For instance, an arithmetic opcode which is shared by a vulnerable trace and a non-vulnerable trace may be replaced with a safe version that checks for overflow. The non-vulnerable trace would revert in the case of an overflow even though the overflow might be benign. Such in-preciseness is an overhead to pay for security in our setting, along with the time and gas overhead. In Section 5.5, we empirically evaluate that the overhead and show that they are negligible.

5.5 Implementation and Evaluation

In this section, we present implementation details of sGuard and then evaluate it with multiple experiments.

5.5.1 Implementation

sGuard is implemented with around 3K lines of Node.js code. It is publicly available at GitHub[1]. It uses a locally installed compiler to compile a user-provided contract into a JSON file containing the bytecode, source-map and abstract syntax tree (AST). The bytecode is used for detecting vulnerability, whereas the source-map and AST are used for fixing the smart contract at the source code level. In general, a source-map links an opcode to a statement and a statement to a node in an AST. Given a node in an AST, sGuard then has the complete control on how to fix the smart contract.

In addition to what is discussed in previous sections, the actual implementation of sGuard has to deal with multiple complications. First, Solidity allows developers to interleave their codes with inline-assembly (i.e., a language using EVM machine opcodes). This allows fine-grained controls, as well as opens a door for hard-to-discover vulnerabilities (e.g., arithmetic vulnerabilities). We have considered fixing vulnerabilities with sGuard (which is possible with efforts). However, it is not trivial for a developer to evaluate the correctness of our fixes as sGuard would introduce opcodes into the inline-assembly. We do believe that any modification of the source code should be transparent to the users, and thus decide not to support fixing vulnerabilities inside inline-assembly.

Second, sGuard employs multiple heuristics to avoid useless fixes. For instance, given an arithmetic expression whose operands are concrete values (which may be the case of the expression is independent of user-inputs), sGuard would not replace it with a function from safe math even if it is a part of a vulnerable trace. Furthermore, since the number of iterations to be unfolded for each loop

[1]https://github.com/reentrancy/sGuard

depends on the number of assignment statements inside the loop, sGuard identifies a number of cases where certain assignments can be safely ignored without sacrificing the soundness of our method. In particular, although we count `SSTORE`, `MSTORE` or `SWAP` as assignment statements in general, they are not in the following exceptional cases.

- A `SWAP` is not counted if it is not mapped to an assignment statement according source-map;
- An assignment statement is not counted if its right-hand-side expression is a constant;
- An assignment statement is not counted if its left-hand-side expression is a storage variable (since dependency due to the storage variables is analyzed regardless of execution order).

In addition, sGuard implements a strategy to estimate the value of memory pointers. A memory variable is always placed at a free memory pointer and it is never freed. However, the free pointer is often a symbolic value. That increases the complexity. To simplify the problem without missing dependency, sGuard estimates the value of the free pointer ptr if it is originally a symbolic value. That is, if the memory size of a variable is only known at run-time, we assume that it occupies 10 memory slots. The free pointer is calculated as $ptr_{n+1} = 10 \times \texttt{0x20} + ptr_n$ where ptr_n is the previous free pointer. If memory overlap occurs due to this assumption, additional dependencies are introduced, which may introduce false alarms, but never false negatives.

Lastly, sGuard allows user to provide additional guide to generate contract-specific fixes. For instance, users are allowed to declare certain variables are

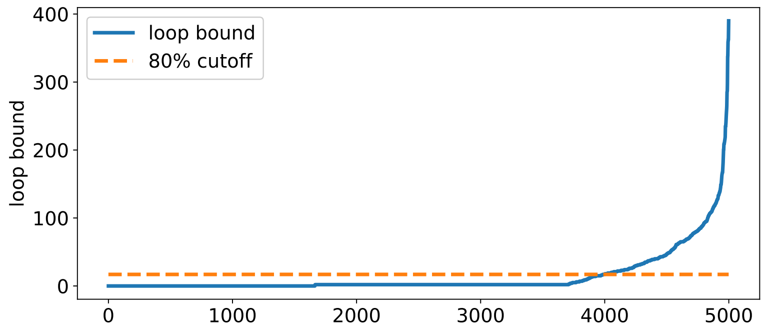


FIGURE 5.9: Loop bounds computed by sGuard

critical variables so that it will be protected even if there is no dependency between the variable and external calls.

5.5.2 Evaluation

In the following, we evaluate sGuard through multiple experiments to answer the following research questions (RQ). Our test subjects include 5000 contracts whose verified source code are collected from EtherScan [80]. This includes all the contracts after we filter 5000 incompilable contracts which contain invalid syntax or are implemented based on previous versions of Solidity (e.g., version 0.3.x). We systematically apply sGuard to each contract. The timeout is set to be 5 minutes for each contract. Our experiments are conducted on with 10 concurrent processes and takes 6 hours to complete. All experiment results reported below are obtained on an Ubuntu 16.04.6 LTS machine with Intel(R) Core(TM) i9-9900 CPU @ 3.10GHz and 64GB of memory.

RQ1: How bad is the path explosion problem? Out of the 5000 contract, sGuard times out on 1767 (i.e., 35.34%) contracts and successfully finish analyzing and fixing the remaining contracts within the time limit. Among them, 1590 contracts are deemed safe (i.e., they do not contain any external calls) and no fix is applied. The remaining 1643 contracts are fixed in one way or another. We note that 38 of the fixed contracts are incompilable. There are two reasons. First, the contract source-map may refer to invalid code locations if the corresponding smart contract has special characters (e.g., copyright and heart emoji). This turns out to be a bug of the Solidity compiler and has been reported. Second, the formats of AST across many solidity versions are slightly different, e.g., version 0.6.3 declares a function which is implemented with attribute *implemented* while the attribute is absent in version 0.4.26. Note that the compiler version declared by *pragma* keyword is not supported in the experiment setup as sGuard uses a list of compilers provided by solc-select [112]. In the end, we experiment with 1605 smart contracts and report the findings.

Recall that the number of paths explored largely depend on the loop bounds. To understand why sGuard times out on 35.34% of the contracts, we record the maximum loop bound for each of the 5000 smart contracts. Figure 5.9 summarizes the distribution of the loop bounds. From the figure, we observe that for 80% of the contracts, the loop bounds are no more than 17. The loop bounds of the remaining 20% contracts however vary quite a lot, e.g., with a maximum of 390. The average loop bound is 15, which shows that the default bounds in existing symbolic execution engines could be indeed insufficient.

RQ2: Is sGuard capable of pinpointing where fixes should be applied? This question asks whether sGuard is capable of precisely identifying where the fixes should be applied. Recall that sGuard determines where to apply the fix based on the result of the dependency analysis, i.e., a precise dependency analysis would automatically imply that the fix will be applied at the right place. Furthermore, control dependency is straightforward and thus the answer to this question relies on the preciseness of the data dependency analysis. Data

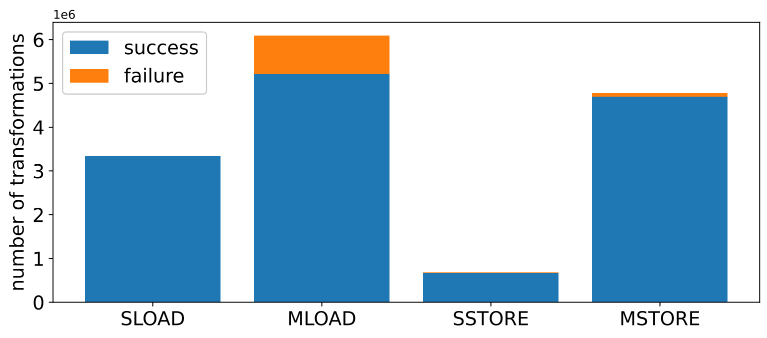


FIGURE 5.10: Memory and storage address transformations

dependency analysis in Algorithm 5 may introduce impreciseness (i.e., over-approximation) at lines 5 and 8 when checking the intersection of reading/writing addresses. In sGuard, the checking is implemented by transforming each symbolic address to a range of concrete addresses using the base address and the maximum offset. The over-approximation is only applied if at least one symbolic address is failed to transform due to nonstandard access patterns. If both symbolic addresses are successfully transformed, we can use the ranges of concrete addresses to precisely check the intersection and there is no over-approximation. Thus, we can measure the over-approximation of our analysis by reporting the number of failed and successful address transformations.

Figure 5.10 summarizes our experiment results where each bar represents the number of failed and successful address transformations regarding the memory (i.e., `MLOAD`, `MSTORE`) and storage (i.e., `SLOAD`, `SSTORE`) opcodes. From the results, we observe that the percentage of successful transformations are 99.99%, 85.58%, 99.98%, and 98.43% for `SLOAD`, `MLOAD`, `SSTORE`, and `MSTORE` respectively. `MLOAD` has the worst accuracy among the four opcodes. This is mainly because some opcodes (e.g., `CALL`, and `CALLCODE`) may load different sizes of data on the memory. In this case, the `MLOAD` may depend on multiple `MSTORE`s, and it becomes even harder considering the size of loaded data is a symbolic value. Therefore, we simplify the analysis by returning true (hence over-approximates) if the size of loaded data is not `0x20`, a memory allocation unit size.

RQ3: What is the runtime overhead of sGuard's fixes? This question is designed to measure the runtime overhead of sGuard's fixes. Note that runtime overhead is often considered as a determining factor on whether to adopt additional checks at runtime. For instance, the C programming language has been refusing to introduce runtime overflow checks due to concerns on the runtime overhead, although many argue that it would reduce a significant number of vulnerabilities. The same question must thus be asked about sGuard. Furthermore, runtime checks in smart contracts introduce not only time overhead but also gas overhead, i.e., gas must be paid for every additional check that is executed. Considering the huge number of transactions (e.g., 1.2 million daily transactions are reported on the Ethereum network [55]), each additional check may potential translate to large financial burden.

To answer the question, we measure additional gas and computational time that users pay to deploy and execute the fixed contract in comparison with the original contract. That is, we download transactions from the Ethereum network and replicate them on our local network, and compare the gas/time consumption of the transactions. Among the 1605 smart contracts, 23 contracts are not considered as they are created internally. In the end, we replicate 6762 transactions of 1582 fixed contracts. We limit the number of transactions for each contract to a maximum of 10 such that the results are not biased towards those active contracts that have a huge number of transactions.

Since our local setup is unable to completely simulate the actual Ethereum network (e.g., the block number and timestamps are different), a replicated

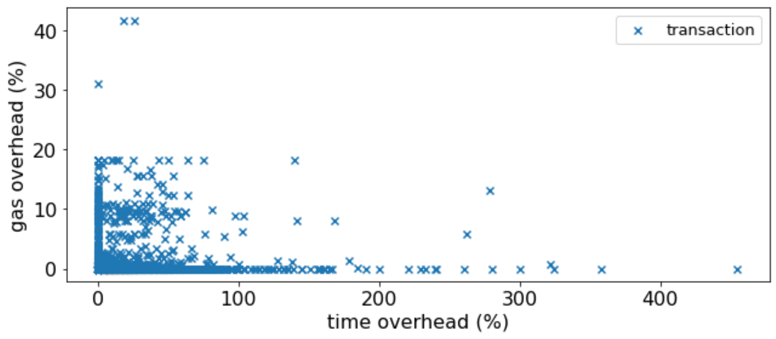


FIGURE 5.11: Overhead of fixed contracts

transaction thus may end up being a revert. In our experiments, 3548 (52.47%) transactions execute successfully and thus we report the results based on them. A close investigation shows that the remaining transactions fail due to the difference between our private network and the Ethereum network except 1 transaction, which fails because the size of the bytecode of the fixed contract exceeds the size limit [113].

Figure 5.11 summarizes our results. The x-axis and y-axis show the time overhead and gas overhead of each transaction respectively. The data shows that the highest gas overhead is 42% while the lowest gas overhead is 0%. On average, users have to pay extra 0.79% gas to execute a transaction on the fixed contract. The highest and lowest time overhead are 455% and 0% respectively. On average, users have to wait extra 14.79% time on a transaction. Based on the result, we believe that the overhead of fixing smart contracts using sGuard is manageable, considering its security guarantee.

For arithmetic vulnerabilities, there is a simplistic fix, i.e., add a check to every arithmetic operation. To see the difference between sGuard and such an approach, we conduct an additional experiment on the set of smart contracts that we successfully fixed (i.e., 1605 of them). We record the total number of bound checks added to the 4 arithmetic instructions (i.e., `ADD`, `SUB`, `MUL` and

Instruction	**sGuard**	**BC**	**BC/sGuard**
`ADD`	576	2245	3.9×
`SUB`	394	2125	5.39×
`MUL`	198	1423	7.19×
`DIV`	179	1508	8.42×

TABLE 5.3: Total number of bound checks

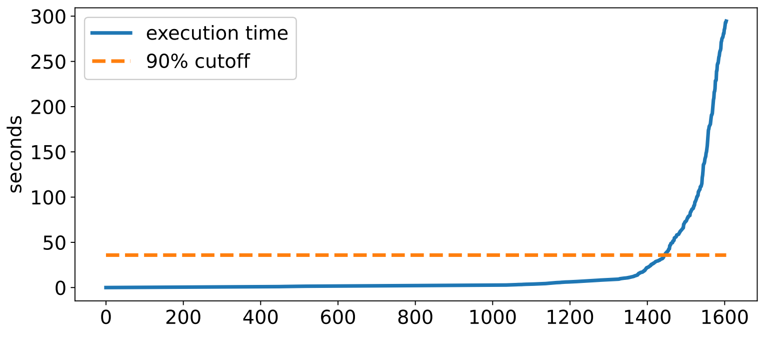


FIGURE 5.12: sGuard execution time

`DIV`) by sGuard and the simplistic approach. The results are shown in Table 5.3, where column BC shows the number for the simplistic approach. We observe that on average sGuard introduces 5.42 times less bound checks than the simplistic approach. Since each bound check costs gas and time when executing a transaction, we consider such a reduction to be welcomed.

RQ4: How long does sGuard take to fix a smart contract? This question asks about the efficiency of sGuard itself. We measure the execution time of sGuard by recording time spending to fix each smart contract. Naturally, a more complicated contract (e.g., with more symbolic traces) takes more time to fix. Thus, we show how execution time varies for different contracts. Figure 5.12 summarizes our results, where each bar represents 10% of smart contracts and y-axis shows the execution time in seconds. The contracts are sorted according to the execution time. From the figure, we observe that 90% of contracts are

No.	Name	#RE	#AR	#TX	Symbolic traces
1	USDT	✗	✗	✗	265
2	LINK	✗	✗	✗	291
3	BNB	✗	✗	✗	128
4	HT	✗	✗	✗	0
5	BAT	✗	✓	✗	128
6	CRO	✗	✗	✗	401
7	LEND	✗	✗	✗	281
8	KNC	✗	✗	✗	443
9	ZRX	✗	✗	✗	0
10	DAI	✗	✗	✗	0

TABLE 5.4: Fixing results on the high profile contracts

fixed within 36 seconds. Among the different steps of sGuard, sGuard spends most of the time to identify dependency (70.57%) and find vulnerabilities (20.08%). On average, sGuard takes 15 seconds to analyze and fix a contract.

Manual inspection of results To check the quality of the fix, we run an additional experiment on the top 10 `ERC20` tokens in the market. That is, we apply sGuard to analyze and fix the contracts and then manually inspect the results to check whether the fixed contracts contain any of the vulnerabilities, i.e., whether sGuard fails to prevent certain vulnerability or whether sGuard introduce unnecessary runtime checks (which translates to considerable overhead given the huge number of transactions on these contracts). The results are reported in Table 5.4 where column RE (respectively AE and TX) shows whether any reentrancy (respectively arithmetic, and tx.origin) vulnerability is discovered and fixed respectively; and the symbol ✓ and ✗ denote yes and no respectively. The last column shows the number of symoblic traces explored.

We observe that the number of symbolic traces explored for three tokens HT, ZRX and DAI are 0. It is because these contracts contain no external calls and thus sGuard stops immediately after scanning the bytecode. Among the remaining 7 tokens, six of them (i.e., LINK, BNB, CRO, LEND, KNC, and USDT) are found to be safe and thus no modification is made. One arithmetic vulnerability in the smart contracts BAT is reported and fixed by sGuard. We confirm that a runtime check is added to prevent the discovered vulnerability. A close investigation however reveals that this vulnerability is unexploitable although it confirms to our definition. This is because the contract already has runtime checks. We further measure the overhead of the fix by executing 10 transactions obtained from the Ethereum network on the smart contract. The result shows that sGuard introduces a gas overhead of 18%. Lastly, our manual investigation confirms that all of the contracts are free of the vulnerabilities.

5.6 Related Work

To the best of our knowledge, sGuard is the first tool that aims to repair smart contracts in a provably correct way.

sGuard is closely related to the many works on automated program repair, which we highlight a few most relevant ones in the following. GenProg [40] applies evolutionary algorithm to search for program repairs. A candidate repair patch is considered successful if the repaired program passes all test cases in the test suite. Dongsun *et al.* presented PAR [41], which improves GenProg by learning fix patterns from existing human-written patches to avoid nonsense patches. Abadi *et al.* automatically rewrites binary code to enforce control flow integrity (CFI) [114]. Jeff *et al.* presented ClearView [115], which learns invariants from normal behavior of the application, generates patches and observes the execution of patched applications to choose the best patch. While there are many other program repair works, none of them focus on fixing smart contracts in a provably correct way.

sGuard is closely related to the many work on applying static analysis techniques on smart contracts. Securify [24] and Ethainter [103] are approaches which leverage a rewriting system (i.e., Datalog) to identify vulnerabilities through pattern matching. In terms of symoblic execution, Luu *et al.* presented the first engine to find potential security bugs in smart contracts [23]. Krupp and Rossow presented teEther which finds vulnerabilities in smart contracts by focusing on financial transactions [27]. Nikolic *et al.* presented MAI-AN, which focus on identifying trace-based vulnerabilities through a form of symoblic execution [28]. Torres *et al.* presented Osiris which focuses on discovering integer bugs [29]. Unlike these engines, sGuard not only detects vulnerabilities, but also fixes them automatically.

sGuard is related to some work on verifying and analyzing smart contracts. Zeus [83] is a framework which verifies the correctness and fairness of smart contracts based on LLVM. Bhargavan *et al.* proposed a framework to verify smart contracts by transforming the source code and the bytecode to an intermediate language called F* [84]. Hirai Yoichi used Isabelle/HOL to verify the Deed contract [116]. M. Fröwis and R. Böhme showed that only 40% of smart contracts are trustworthy based on their call graph analysis [88]. Chen *et al.* showed that most of the contracts suffer from some gas-cost programming patterns [89].

Finally, sGuard is remotely related to approaches on testing smart contracts. ContractFuzzer [26] is a fuzzing engine which checks 7 different types of vulnerabilities. sFuzz [43] is another fuzzer which extends ContractFuzzer by using feedback from test cases execution to generate new test cases.

5.7 Conclusion

In this work, we propose an approach to fix smart contracts so that they are free of 4 kinds of common vulnerabilities. Our approach uses run-time information and is proved to be sound. The experiment results show the usefulness of our approach, i.e., sGuard is capable of fixing contracts correctly while introducing only minor overhead. In the future, we intend to improve the performance of sGuard further with optimization techniques.

Chapter 6

SPAIR: Towards Repairing Smart Contract Specification

Specifications are indispensable to formal verification of the functional correctness of programs and are typically assumed correct. However, writing proper specifications is time-consuming and error-prone, thus specifications written by humans can be buggy. In this work, we propose a novel approach to repair specifications for smart contracts by leveraging abductive inference and constraint solving. Our approach has been implemented in a tool, namely SPAIR, which automatically discovers inconsistencies between the specification and the implementation, and then generates recommendations for repairing specifications. Experiments on 10 high-profile smart contracts show that SPAIRis able to recommend all the desired specifications. The repairing process is not limited to specifications that contain a single bug but also handles specifications with multiple bugs efficiently within a minute in most cases.

6.1 Introduction

Smart contracts are programs that run on top of blockchain. They are often used to implement financial applications and increasingly other critical applications. A bug in a smart contract thus could result in a massive loss of valuable digital assets, which has been demonstrated time and time again [73], [90]. More importantly, due to the immutability of blockchain (which is one of its fundamental properties), a smart contract cannot be patched once it is deployed. In other words, once deployed, a bug in the smart contract would make it forever vulnerable. Thus, it is vital to guarantee the correctness of smart contracts before they are deployed.

Built upon the idea that different contracts have different correctness specifications, recently, several approaches have been proposed to support the falsification or verification of manually specified correctness specifications. VerX [30] relies on predicate abstraction to verify temporal properties of smart contracts. SmartPulse [31] transforms smart contracts containing temporal properties into the intermediate representation of the Boogie verifier [117]. Solc-verify [32] also leverages Boogie to verify its function-level specifications such as preconditions, postconditions, and loop invariants. These existing approaches work under a common assumption that specifications are perfect while the implementation may contain errors. Consequently, developers are responsible for writing code that aligns with given specifications. In practice, since the specifications are written by humans as well, a more likely scenario is that specifications could be buggy (at least initially) as well. A research question that has been overlooked so far is thus: *how to debug and repair a specification if it turns out to be buggy?* It is worth mentioning that maintaining correct specifications is crucial for smart contracts as they not only help to discover implementation bugs but also play a key role in the correctness certification.

In this work, based on the assumption that specifications can be buggy as well, we propose a novel approach to automatically repair them. Repairing a specification is challenging as the specification of a function is dependent on other functions (through function calls) and there may be many ways of repairing the specification. We propose SPAIR, the first approach that automatically discovers inconsistency between the specifications and the implementation, and generates recommendations for repairing the specifications. SPAIR is useful in various scenarios. First, while writing specifications along with the implementation, SPAIR can suggest absent conditions. For example, given an implementation of a function and its postcondition, SPAIR can quickly generate a correct precondition, providing developers insights into the type of conditions the function anticipates. That is, SPAIR often suggests the weakest precondition, any condition that is logically stronger than our suggestion is considered valid. Second, when a hot-fix is introduced to address a vulnerability, the specification needs to be updated accordingly. In this case, SPAIR can be applied to patch the specifications of the affected functions. This automated process saves developers significant time and effort that would be spent on manually identifying and repairing the affected specifications.

Given a smart contract with potentially buggy specifications (in the form of a precondition/postcondition for each function), SPAIR works in three steps. Initially, it creates a call graph from the smart contract, in which each node is a function and there is a directed edge from the caller to the callee. The graph is then divided into connected components. Subsequently, SPAIR patches the specifications in a bottom-up manner by identifying connected components containing functions associated with incorrect specifications. Lastly, SPAIR ranks all generated patches and selects the best one for each function. The bottom-up approach guarantees that all the called functions are patched before the caller function. It is adopted because SPAIR is built on top of a functional verification engine, where the correctness of the caller function relies on the correctness of the called functions. By systematically patching the specifications in a bottom-up manner, SPAIR ensures that the entire connected component is properly verified and their dependencies are correctly maintained.

To summarize, we make the following main contributions:

- We propose a novel approach to identify buggy specifications and patch them in a provably correct manner.
- We implement our approach as a self-contained tool SPAIR for patching specifications of Solidity smart contracts.
- We thoroughly evaluate SPAIR on 10 high-profile smart contracts. The experiment shows that SPAIR can effectively infer 100% missing preconditions and postcondition within 7 seconds. Furthermore, it can patch specifications with more than one bug efficiently (within 77 seconds) as well.

Outline. In Sections 6.2 and 6.3, we provide some background about smart contracts and illustrate how our approach works through examples. The problem is then defined formally in Section 6.4. In Section 6.5, we present the details of our approach. The experiment results are presented in Section 6.6. We discuss related work in Section 6.7 and conclude this work in Section 6.8.

6.2 Specification Language

In view of diverse vulnerabilities that have been discovered in Solidity smart contracts and the fact that it is insufficient to focus on specific bugs, several approaches [30], [31], [102] have been developed recently to support user-provided correctness specification. There are many approaches that support different smart contract specifications such as linear temporal logic in VerX [30] or predefined properties in VeriSmart [66]. Without loss of generality, in this work, we focus on a specification language based on the classical Floyd-Hoare preconditions and postconditions, a widely used specification language [32], [117]. Note that this work relies on iContract as its foundation. The detailed formalization of our specification language is discussed in Section 4.3.2. In the following, we present the necessary formalization of our specification language so that we can explain precisely how our approach works.

Note that other forms of specification such as contract invariants [32] and call invariants [118] can be naturally supported by our approach. Our specification language is constituted of predicates defined using the syntax below.

$$\begin{array}{rcl}
\Phi, p, q & := & \Psi \mid \Phi \vee \Phi \\
\Psi & := & a \otimes a \mid \Psi \wedge \Psi \\
a & := & e \mid a \oplus a \mid \odot a \\
e & := & l \mid v \mid v[a] \mid v.m \mid old(v) \mid g(v)
\end{array}$$

In general, a predicate Φ is a disjunction with one or multiple conjunctions Ψ. Each conjunct in Ψ is a relational predicate using a relational operator $\otimes$ (i.e., $>$, $\geq$, $=$, $\neq$, $<$, $\leq$). The left-hand side and right-hand side of a relational predicate are arithmetic expressions a. An arithmetic expression may have one atomic expression or multiple of them connected by binary operators $\oplus$ (i.e., $+$, $-$, $*$,

/) or unary operators $\odot$ (i.e., $\neg$, $-$). An atomic expression can be a literal l, a variable v, a member access $v.m$, and an index access $v[a]$. The expression $v.m$ accesses the value stored in the member m of a struct v, whereas the expression $v[a]$ accesses the value at key a of a mapping v.

The semantics is defined according to a satisfaction relation $S, H \models \Phi$ which is defined in the standard way, as shown in Section 4.3.2. Next, we define how the constructs in our specification language are defined. Regarding function invariants, given a function $m(\overline{v}) = s$ associated with multiple ensures(p, q) statements, the function is correct iff for each ensures(p, q) statement, the following is satisfied:

$$\forall \sigma, \sigma'.\ \sigma \models p \land (s, \sigma) \rightsquigarrow^* (\text{skip}, \sigma') \implies \sigma' \models q.$$

Intuitively, for every state σ that satisfies the precondition p, the final state σ' after executing the body statement s from the state σ should also satisfy the postcondition q.

6.3 Motivation Example

In this section, we motivate our approach and illustrate its underlying idea through two examples.

Example 6.3.1 (Suggesting missing conditions)**.** During the process of developing a function specification, it is typical for developers to first write either the precondition or postcondition and then proceed to the other. In this scenario, we demonstrate how our approach can assist developers in writing a correct

```
 1 @ensures(
 2 ...,
 3 balances[msg.sender]==old(balances[msg.sender])-_value
 4 && balances[_to]==old(balances[_to])+_value
 5 )
 6 function transfer(address _to, uint256 _value)
 7 returns(bool success) {
 8   if (balances[msg.sender] >= _value && _value > 0) {
 9     balances[msg.sender] -= _value;
10     balances[_to] += _value;
11     emit Transfer(msg.sender, _to, _value);
12     return true;
13   } else {
14     return false;
15   }
16 }
```

FIGURE 6.1: A function with a missing precondition. The specification is fixed by adding a precondition (line 2) such that the postcondition is satisfied

and general specification. Figure 6.1 shows the underlying logic of the function `transfer`, which is extracted from the smart contract named `BAT`[1]. This function allows token transferring between two users. The postcondition specifies that the balance of the sender and receiver increases and decreases with the same amount respectively. The question is: *what is the precondition that can make the specification correct?* A common mistake is to have the following answer:

```
balances[msg.sender] >= _value && _value > 0.
```

This precondition ensures that only the true branch of the if-statement at line 8 is executed. Although the precondition may seem reasonable, it is incorrect. Considering the case where `sender` and `receiver` are the same person (i.e., `msg.sender == _to`), the postcondition is violated when `_value > 0`. In contrast, relying on abductive inference [119], our approach suggests a general precondition as follows:

[1]https://etherscan.io/token/0x0d8775f648430679a709e98d2b0cb6250d2887ef

```
(balances[msg.sender] >= _value && msg.sender != _to)
|| (_value == 0)
```

Intuitively, we can split the precondition into two scenarios. The first scenario indicates that if `_value` $\neq$ `0`, the transfer should not be self-transfer, meaning `sender` and `receiver` should be different (i.e., `msg.sender!=_to`). The second scenario indicates that zero-value token transfer is allowed (i.e., `_value==0`). It is noted that zero-value token transfer with complicated transfer fee or none-zero-value token self-transfer have been identified as the root cause of various bugs, including price oracle manipulation [120], [121]. Thus, such cases should be avoided. In summary, our approach can suggest general and correct preconditions for functions.

#	Function	Patch	Type	Correct
1	`ER._transfer`	`true`	P-patch	✗
2	`ER.transferFrom`	`true`	P-patch	✗
3	`T2.transferFrom`	`_amount < 1`	P-patch	✓
4	`ER._transfer`	`old(balances[_sender]) > _amount && ...`	Q-patch	✗
5	`ER.transferFrom`	`balances[_sender] == old(balances[_sender]) - _amount && ...`	Q-patch	✗
6	`T2.transferFrom`	`allowance[_sender][msg.sender] == old(allowance[_sender][msg.sender]) - _amount && ...`	Q-patch	✓
7	`ER._transfer`	`_amount >= 0;true`	F-patch	✗
8	`ER.transferFrom`	`_amount >= 0 && _recipient != 0; true`	F-patch	✗

TABLE 6.1: Patches generated by SPAIR to repair the specifications in Figure 6.1. We use `T2` and `ER` to refer to contract `TetherToken` and its direct parent `ERC20Upgradeable`. The precondition and postcondition of F-patch are separated by a semicolon

Example 6.3.2 (Patching existing specifications). Figure 6.2 shows changes made to address a bug in Tether Gold token (XAUt), which has the marketcap of around 473 million USD. The bug was discovered by Blocksec on May 27th

```
1  contract ERC20Upgradeable {
2    ...
3    @ensures(...)
4    function _transfer(...) {...}
5    @ensures(...)
6    function transferFrom(...) {...}
7  }
8  contract TetherToken is ERC20Upgradeable {
9    ...
10   @ensures(
11     isTrusted[_recipient] && _amount >= 0, // pre
12     old(allowance[_sender][msg.sender]) == allowance[_sender][msg.
           sender] // post
13   )
14   function transferFrom(
15     address _sender,
16     address _recipient,
17     uint256 _amount) public returns (bool)
18   {
19     ...
20     -  if (isTrusted[_recipient]) {
21     -    _transfer(_sender, _recipient, _amount);
22     -    return true;
23     -  }
24     return super.transferFrom(_sender, _recipient, _amount);
25   }
26 }
```

FIGURE 6.2: A bug in contract `TetherToken` at lines 20-23. It is fixed by simply deleting vulnerable codes at lines 20-23

2023 [122]. It lies in the if-statement at lines 20-23. In particular, this statement allows an attacker to transfer XAUt to a trusted recipient. Although it is not possible to directly benefit from the bug, the attacker can transfer XAUt to manipulate the price of XAUt in the WETH-XAUt pool and make profit. The specification at lines 10-13 states that if the recipient is trusted then the allowance is unchanged.

The bug is repaired by removing codes from line 20 to 23. However, this results in an inconsistency between the repaired code and the existing specification. Therefore, the specification must be updated accordingly. Instead of manually modifying the specification, which is a time-consuming and error-prone process, SPAIR is able to patch the specification in a provably correct manner. Given the repaired code, Table 6.1 shows 8 patches produced by SPAIR. The *#Function* column combines the contract name and function name to differentiate between identical function names in various contracts. The *Correct* column indicates that the patch is correct (i.e., ✓) or incorrect (i.e., ✗) where correct patch means that all specifications are verified after applying the patch. The table consists of 3 types of patches:

- *Precondition* (P-patch): replacing the precondition of a function with an inferred precondition. For example, patch #3 replaces the precondition of the function `T2.transferFrom` with an inferred condition `_amount<1`.
- *Postcondition* (Q-patch): replacing the postcondition of a function with an inferred postcondition. For example, the patch #6 replaces the postcondition of the function `T2.transferFrom` with the inferred condition shown in Table 6.1.
- *Function call* (F-patch): replacing both the precondition and postcondition of a function with new precondition and new postcondition, which are inferred from the call site. For example, patch #7 replaces both precondition and postcondition of the function `ER.transferFrom`. This patch is inferred from the function call at line 24 of Figure 6.2.

This example shows that there may be many ways of correctly patching the specification. The high-level principles of SPAIR are that (1) it suggests correct specification repair that requires minimal change (based on Occam's Razor principle) and (2) it always keeps the programmer in the loop. For example, a straightforward solution is to patch specifications one by one until all specifications become correct. However, it may not be practical as it could result in significant changes to the existing specifications. We constraint the search space by replacing precondition or postcondition of a function with an appropriate inferred component (i.e., precondition or postcondition). As a result, a precondition patch only replaces precondition of a function and a postcondition patch only replaces the postcondition of a function.

Based on the above-mentioned principles, we thus rank the generated patches according to a certain distance function which measures the magnitude of the change and suggests top-ranked one to the programmer. SPAIR uses Graph Edit Distance (GED) metric to select the best patch. Out of the 8 patches available, patch #3 is chosen because it has a minimal distance to the original specifications and guarantees the correctness of all the specifications.

6.4 Detailed Approach

SPAIR works in three main phases including (1) function-level verification, (2) patch generation, and (3) patch ranking. Given a smart contract with potentially buggy specifications, the first phase identifies functions whose specifications are buggy. In the second phase, SPAIR generates a limited number of patches. Finally, in the third phase, the generated patches are ranked based on the GED distance. In the following, we introduce a high-level overview of our approach and then present details of each phase.

6.4.1 High-level Algorithm

The overview of our patching approach is shown in Algorithm 7. From a high-level point of view, it works as follows. It first builds the call graph of the smart constract which is then divided into connected components (line 3). Each connected component is then topologically sorted (line 6) to determine the patching order. Second, SPAIR generates a top-ranked patch for each connected component (lines 4-8). Subsequently, it merges all the top-ranked patches into a final patch (line 7). We assume that there are no mutual recursive function calls and thus the topological sort always has a solution. The assumption is reasonable considering writing mutual recursive function calls is discouraged in smart contracts to avoid denial of service (DoS) attacks [123].

In the $update$ function (lines 10-24), if a function is selected (line 15), the algorithm generates a P-patch (line 20) or Q-patch (line 21). For each function call made by function m, it is possible to overwrite precondition and/or postcondition of the callee with an F-patch (line 22).

It is possible that SPAIR is unable to infer conditions due to the limitation of the underlying patch generation engine. We always set a time limit for each patch generation. To ensure the correctness of an F-patch, derived from a specific call site, we include a validation step to verify the F-patch against the code of the invoked function.

6.4.2 Function Verification

SPAIR targets function-level correctness of smart contract. Each function is encoded and verified separately. The correctness of a function is defined in terms of function encoding as follows.

Definition 7 (Function-level correctness)**.** The encoding function $post(\sigma_i, s_i)$ takes a pre-state σ_i and a statement s_i as inputs and produces post-state σ_k as output. Given a function m with implementation s, a specification ensures(p, q) is correct if $post(p, s)$ returns σ such that $\sigma \Rightarrow q$.

Algorithm 7: Patch Generation

```
/* Entry point                                                    */
def main(...):
    let $result \leftarrow \emptyset$ ;
    let $ccs \leftarrow connected\_components(call\_graph)$;
    for $cc \in ccs$ do
        global $patches \leftarrow \emptyset$ ;
        $update(toposort(cc), \emptyset)$ ;
        $result \leftarrow result \cup top(patches)$ ;
    end
    return $result$ ;
/* Update patch                                                   */
def update(methods, patch):
    if $methods$ is empty then
        add $patch$ to $patches$ ;
        return;
    end
    let $m \leftarrow methods.pop()$;
    // No patch
    if $is\_verified(m)$ then
        $update(methods, patch)$;
        return;
    end
    // P-patch
    $update(methods, patch \cup \{P\text{-}patch(m, patch)\})$;
    // Q-patch
    $update(methods, patch \cup \{Q\text{-}patch(m, patch)\})$;
    // F-patch
    for $c \in m.internal\_calls$ do
        $update(methods, patch \cup \{F\text{-}patch(c, patch)\})$;
    end
```

While encoding a function, we have a dedicated procedure to handle assertion failures. That is, if an assertion is failed, the encoding process terminates immediately. For example, when evaluating $post(\sigma_i, assert(e))$, an exception is thrown if the condition $\sigma_i \Rightarrow e$ is not satisfied. Conversely, if the condition is satisfied, the output remains the same, i.e., the state σ_i. The encoding of the different statements in a smart contract is straightforward and will not be detailed.

However, the complicated rules revolve around describing the interaction between different functions. That is, if there exists an interaction (e.g., internal calls and external calls) from the current function to other functions, we rely on their specifications to produce the encoding. In such cases, these specifications are assumed to be correct. Every interaction is encoded as follows.

Definition 8 (Modular Verification)**.** Given a function m with a specification ensures(p, q), then $post(\sigma_i, m(\overline{e}))$ returns $\sigma_i \wedge q$ if $\sigma_i \Rightarrow p$

The encoding process terminates if the precondition of a specification is not satisfied by the current encoding context (i.e., $\sigma_i \not\Rightarrow p$). Additionally, we implicitly handle global variable modifications while invoking a function call. Given a specification ensures(p, q) of a function call $m(\overline{e})$, m modifies a global variable g, but this modification is not included in the postcondition q. It is clear that q is correct but insufficient, potentially leading to incorrect conclusions. To address this problem, we analyze the body of m and record any modifications made to global variables. Two variables g_{old} and g_{new} are used to represent the variable g where g_{old} refers to g in the precondition, while g_{new} refers to g in the postcondition.

6.4.3 Patch Generation

SPAIR generates specification repair candidates in a bottom-up manner. This aims to identify the patching order in the way that a function is examined at most once.

To derive the patching order, SPAIR creates a call graph from smart contract, in which each node is a function and there is a directed edge from the caller to the callee. Although the call graph is a directed graph, when determining connected components, we treat the graph as an undirected one. To minimize the

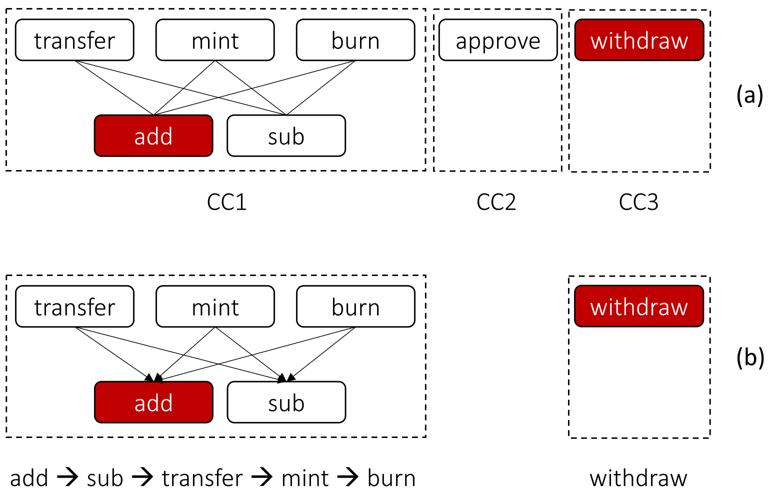


FIGURE 6.3: Connected components and patching orders

overhead associated with patching irrelevant functions, SPAIR only patches connected components that have functions associated with buggy specifications. By examining the call graph, we can identify zero or multiple connected components. Generating a patch for a contract is equivalent to producing individual patches for each connected component and subsequently merging them together (refer to line 7 of Algorithm 7). For example, the graph in Figure 6.3 (a) has 3 connected components $CC1$, $CC2$, and $CC3$. Two functions, namely `add` and `withdraw`, have buggy specifications. Because `add` belongs to $CC1$ and `withdraw` belongs to $CC2$, only specifications of functions in $CC1$ and $CC3$ are patched. According to the directed graph in Figure 6.3(b), SPAIR performs topological sort to determine the order of patching, as illustrated below the graph.

As discussed before, to constraint the solution space, we focus on three types of patches: precondition patch (P-patch), postcondition patch (Q-patch), and function call patch (F-patch). These patches are inferred from the code. Our inference process always follows the rule that all specifications except the one we are trying to infer are correct. This is because, without this assumption, our inference process may fall into a loop. The correctness of a patch is defined as follows.

Definition 9 (Patch correctness)**.** Given a set of original specifications I, a patch R substitutes specifications $I_o \subset I$ with I_n and results in $I' = I_n \cup I \setminus I_o$. The patch R is correct iff every function is correct with respect to its specification after the substitution.

There is an infinite number of preconditions or postconditions that can be used to construct a correct patch. For example, the condition $x > 10$ is semantically equivalent to $x > 10 \vee x > 20$ or $\neg(x \leq 10)$. Although various variations can be used to produce correct patches, they are less useful. Therefore, whenever feasible, we aim to generate the weakest preconditions and strongest postconditions to further reduce the number of less useful patches.

Definition 10 (P-patch)**.** Given a function m associated with an ensures(p, q). A P-patch substitutes p with the weakest precondition p' to construct a correct specification ensures(p′, q).

Finding the weakest precondition has been widely studied. In this work, we combine abductive inference [119] with temporary variable elimination to compute the weakest precondition q'. Formally,

$$p' \equiv \text{QE}(\forall V_p.post^*(true, s) \Rightarrow q) \wedge p''$$

where s is the implementation of the function m; V_p includes variables that are invisible to the precondition such as local variables, temporary variables; p'' is introduced to guarantee that all assertions are satisfied. The procedure QE performs variable eliminations. The encoding $post^*(true, q)$ implies that we start the encoding process with an empty precondition (i.e. precondition is

$true$). The main difference between $post$ and $post^*$ is that $post$ performs assertion validations, while $post^*$ does not. For example, given a statement $assert(e)$, the encoding function $post$ terminates if $\sigma \not\Rightarrow e$, but $post^*$ skips the validation step. Note that p' always exists. In the worst-case scenario, p' is $false$.

We employ the rules shown in Figure 6.4 to compute p''. The default value of p'' is $true$. The purpose of introducing p'' is to ensure that all the assertions within the function are satisfied. Each encoding rule is of the form

$$\frac{premise_0 \ldots premise_i}{p''_i, s_i \rightsquigarrow p''_k}$$

This transition rule means given a condition p''_i, a statement s_i, it executes $premise_0$, ..., $premise_i$ to obtain the condition p''_k.

$$\frac{x = \text{QE}(\forall V_p.\sigma \Rightarrow e)}{p'', assert(e) \rightsquigarrow p'' \wedge x}\ \textsc{Assert} \qquad \frac{x = \text{QE}(\forall V_p.\sigma \Rightarrow p)}{p'', \text{ensures}(\text{p, q}) \rightsquigarrow p'' \wedge x}\ \textsc{Modular}$$

$$\frac{p''_1, s_1 \rightsquigarrow p''_1 \qquad p'', s_2 \rightsquigarrow p''_2}{p'', \text{if b then } s_1 \text{ else } s_2 \rightsquigarrow p''_1 \vee p''_2}\ \textsc{If} \qquad \frac{x = \text{QE}(\forall V_p.post(\sigma \wedge e \wedge b, s) \Rightarrow e)}{p'', linv(e); \text{while } b \text{ do } s \ \rightsquigarrow p'' \wedge x}\ \textsc{Loop}$$

$$\frac{p'', s_1 \rightsquigarrow p''_1 \qquad p''_1, s_2 \rightsquigarrow p''_2}{p'', s_1; s_2 \rightsquigarrow p''_2}\ \textsc{Sequence}$$

FIGURE 6.4: Rules to compute p''

Intuitively, rules ASSERT and MODULAR strengthen p'' by adding a new constraint x. Rules IF and SEQUENCE show the process of accumulating p''. Rule LOOP generates a new constraint x for the loop invariant linv(e).

```
@ensures(true, z > 10)
function main(uint x) {
  assert(x > 1);
  uint v = x - 1;
  z = v - z;
}
```

For example, the code above shows a public function $main$, which has a global variable z, a parameter x, and a local variable v. The specification ensures(true, z > 10) is falsified. The weakest precondition p' is computed as follows,

$$\begin{aligned} p' &\equiv \text{QE}(\forall t_0, t_1.(t_0 = x - 1 \wedge t_1 = t_0 - z \Rightarrow t_1 > 10)) \wedge x > 1 \\ &\equiv x > 11 + z \end{aligned}$$

Note that the encoding process undergoes a post-processing step to correctly align variables to the context. For example, although the variable z is assigned a new value in the given example, in the encoding, z refers to the variable that is not assigned.

Definition 11 (Q-patch)**.** Given a function m associated with an ensures(p, q). A Q-patch substitutes q with the strongest postcondition q' to construct a correct specification ensures(p, q′).

Similar to P-patch, the process of finding the strongest postcondition also depends on variable elimination. Formally,

$$q' \equiv \text{QE}(\exists V_q.\ post(p, s))$$

where V_q includes variables that are invisible to the postcondition such as local variables, and temporary variables. Note that our specification supports a utility function old(), where old(x) refers to the value of x at the beginning of the function. If a variable v is updated, it has two versions old(v) and v, both old(v) and v can be used to constitute the postcondition. Note that q' may not exist due to failed assertions.

```
@ensures(x + z >= 9, false)
function main(uint x) {
  uint v = x - 1;
  z = v - z;
}
```

For example, the code above shows the falsified specification ensures($x + z \geq 9$, false). The strongest postcondition q' is computed as follows,

$$\begin{aligned} q' &\equiv \text{QE}(\exists t_0.\ (x + old(z) \geq 9 \land t_0 = x - 1 \land z = t_0 - old(z))) \\ &\equiv z = x - old(z) - 1 \end{aligned}$$

Definition 12 (F-patch)**.** Given a function m associated with an ensures(p, q). An F-patch substitutes the precondition p and the postcondition q with p' and q' respectively, where the precondition p' and the postcondition q' are inferred from its call site to construct a correct specification ensures(p′, q′)

To construct an F-patch for a function m, we first identify the location where the call is made, and subsequently, we infer both precondition and postcondition for m. Let us assume that a function f has the implementation $s = \{s_0; m(\overline{e}); s_1;\}$ where $m(\overline{e})$ is the invocation of m, s_0 and s_1 are statements. Given the specification ensures(a, b) associated with f, we again rely on variable elimination to infer the precondition p' and the postcondition q'. Formally,

$$p' \equiv \text{QE}(\exists V_0.post^*(a, s_0)) \land p'';\ q' \equiv \text{QE}(\forall V_1.post(a, s) \Rightarrow b)$$

where V_0 and V_1 consist of variables that are invisible to p' and q' respectively. Because we are inferring a specification for m, the specification of m is not used as in Definition 2. We add auxiliary variables to represent parameters and returned variables of m. For example, a function add has two parameters

x, y, and a return variable r. The encoding of a statement $v = add(10, 20)$ is $c_0 = 10 \wedge c_1 = 20 \wedge c_2 = v$ where c_0, c_1 and c_2 are placeholders representing x, y and r respectively. Through variable elimination, we derive $c2 = c0 + c1$. Subsequently, we perform a substitution to obtain a valid specification $r = x+y$.

```
 1 @ensures(true, z == x - old(z))
 2 function main(uint x) public {
 3   require(x > z);
 4   z = subtract(x, z);
 5 }
 6
 7 @ensures(true, c == a + b)
 8 function subtract(uint a, uint b) public
 9 returns(uint c) {
10   assert(a >= b);
11   c = a - b;
12 }
```

For example, the code above shows two functions including $main$ and $subtract$, where z is a global variable. By analyzing the function call at line 4, the precondition p' and postcondition q' of $subtract$ are computed as follows.

$$\begin{aligned}
p' &\equiv \mathrm{QE}(\exists x.(a = x \wedge b = z \wedge x > z)) = (a > z \wedge b = z) \\
q' &\equiv \mathrm{QE}(\forall x.(a = x \wedge b = z \wedge x > z \wedge c = z \Rightarrow z = x - old(z))) \\
&\equiv (c = a - old(z))
\end{aligned}$$

After having p' and q', we re-verify them against the body of the function $subtract$ and confirm that they are correct.

6.4.4 Patch Ranking

Based on the assumption that a patched specification is likely syntactically close to the original one [124], we propose an approach that measures the syntactic distance between a candidate solution and the original specification. We measure the syntactic difference by looking at the difference between ASTs using Graph Edit Distance (GED). The GED algorithm is chosen as it is well-suited for comparing ASTs. In essence, GED measures the minimal number of actions such as insertion, deletion, and substitution needed to transform the candidate solution AST to the original specification AST. Given two expressions, we convert them into symbolic expressions [125], which are then represented in a tree data structure, in which each node is either an operator or an operand and there is a directed edge from the operator to the operand. Next, we employ an existing GED implementation called Networkx [126] to compute the difference between them. For example, recall the precondition suggestions in Table 6.1, the tree structure representations of #pre (original precondition), #sug1 (first suggestion), and #sug2 (second suggestion) are shown in Figure 6.5. Their graph edit distance, denoted by GED, are as follows,

- GED$(\#pre, \#sug1) = 2$. To obtain #pre from #sug1, we substitute nodes $isTrusted$ and $select$ with 0 and $\neq$ respectively.
- GED$(\#pre, \#sug2) = 8$. To obtain #pre from #sug2, we need to add 4 edges and 4 nodes.

It is noted that each operation has a cost of 1. We prefer #sug1 over #sug2 because GED$(\#pre, \#sug1)$ is smaller than GED$(\#pre, \#sug2)$.

Since a patch may substitute more than one condition (i.e., precondition and/or postcondition). The distance between a patched specification and the original specification is determined by the sum of the distances of all substituted conditions. Formally, it is computed as follows.

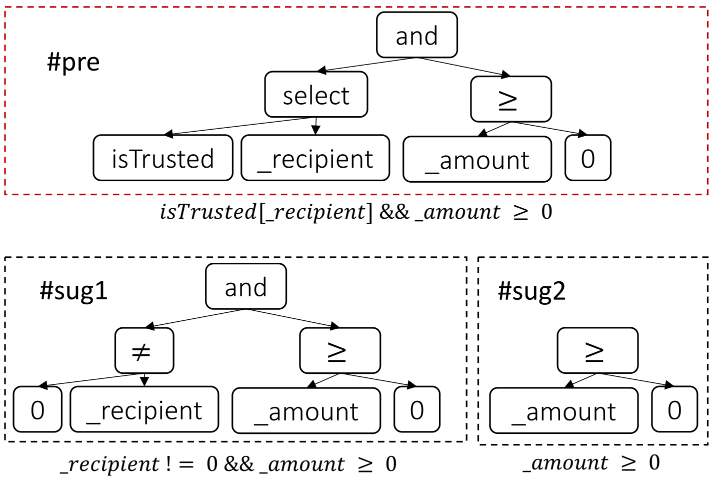


FIGURE 6.5: An example of GED

Definition 13. Given a set of original (pre/post) conditions I, a patch R is a map that maps a predicate $i \in I$ with its substitution i' then GED$(R, I) = \sum_{(i,i')\in R}$ GED(i, i').

We choose a top-ranked patch, which has the smallest distance, as the solution. In the example above, #sug1 is chosen.

6.5 Implementation

In this section, we elaborate on the implementation details of our tool. SPAIR consists of more than 1k lines of Python code. It is built on top of Slither [127], a popular static analysis tool for Solidity. SPAIR directly consumes SlithIR [127] to produce the encoding of functions, as well as correctness specification as shown in Section 6.2. Most of the Solidity features are supported in a straightforward way. We discuss some of the non-trivial implementation details here.

Tactic: Tactic is a procedure that guides the z3-solver [128] in attempting to find a solution to a given problem. It can perform various operations, such as simplification, rewriting, splitting, etc. A tactic often provides a general solution to a problem. We have to chain multiple tactics together to achieve the goal. Quantifier elimination is frequently used in SPAIR to produce repair candidates. However, it often returns overly complicated candidates that are not human-friendly. For example, instead of returning $balances[x] \geq 0$, the solver may return $store(balances, x, 0)[x] \geq 0$. Although the semantics is the same in both cases, the second one is less readable. We apply different optimizations to eagerly simplify repair candidates. For example, to obtain the simplified repair candidate $balances[x] \geq 0$ from the complicated one $store(balances, x, 0)[x] \geq 0$, we first eagerly replace all $store(\dots)[\dots]$ term by an *if-then-else* term. After that, we eliminate all *if-then-else* term using tactic *elim-term-ite*

Quantifier elimination is an important part of our work. It has some well-known limitations. For instance, the quantifier engine is not optimized for quantifiers over arrays, and it may occasionally fail to terminate. To address this challenge, we make an effort to minimize the introduction of extra temporary variables. The reason is although introducing extra variables may enhance the overall versatility of the implementation, it can potentially disrupt the quantifier elimination process.

Variable: A Solidity function consists of three distinct variable types including parameter variables $\overline{P}$, local variables $\overline{L}$, and state variables $\overline{S}$. Formally, $\overline{V} = \overline{P} \times \overline{L} \times \overline{S}$. Parameter variables refer to variables that are declared in the parameters of the function. Local variables are temporary variables used within the scope of the function. State variables encompass globally declared variables and predefined variables such as $msg.sender$, $msg.value$, and now. A variable v could be presented by multiple temporary variables $c_0, \dots, c_k \in \overline{T}$. Given an encoding σ, the variable v in σ is v that is evaluated at the end of the function. Whereas, the temporary variable c_0 is v that is evaluated at the beginning of the function. We identify variables to be eliminated differently in different elimination processes. For example, $\overline{V_p} = \overline{L} \times \overline{T}$ and $\overline{V_q} = \overline{L} \times \overline{T}$.

Recursion in F-patch: To compute an F-patch for the function call $m(\overline{e})$ called in $f(\overline{v})$, the specifications of any other functions called in $f(\overline{v})$ are required. However, there exists a scenario such that another function call $m(\overline{e'})$ occurs. In this case, the specification of m cannot be used to represent $m(\overline{e'})$. To solve this problem, the encoding of $m(\overline{e'})$ is used. For other function calls that are not m, we employ their specifications.

Inline Assembly To provide developers flexibility in terms of optimizing the implementation, assembly can be used in Solidity smart contracts. However, it also opens doors to many complicated bugs [129]. All the verifiers (such as VeriSmart [66], SmartPulse [31]) that work on the source code instead of bytecode, are unable to precisely handle inline assembly. In our current implementation, we only print out warning messages and ignore inline assembly.

6.6 Evaluation

In this section, we evaluate SPAIR through multiple experiments. The experiments are designed to answer the following key research questions (RQ). Each RQ aligns with a particular scenario where SPAIR demonstrates its potential usage.

RQ1 *Is it possible for* SPAIR *to infer missing specifications?*

Project	#Cont	#Func	#Cond	#Spec	#Line	#Tran (mil)
BAT	4	16	20	16	179	3.97
BNB	2	13	22	13	150	1.00
HT	4	13	4	13	127	0.67
HOT	3	22	29	22	279	0.95
IOTX	8	32	28	32	500	0.28
QNT	5	24	13	24	239	1.21
MANA	11	28	21	28	282	2.50
ZIL	9	35	42	35	353	0.44
NXM	3	37	36	37	448	0.12
SHIB	4	33	12	33	448	9.50

TABLE 6.2: Statistics of 10 high-profile projects. For each project, the table shows the number of contracts (#Cont), functions (#Func), conditional statements (#Cond), specification statements (#Spec), line of codes (#Line), and transactions (#Tran)

RQ2 *Can* SPAIR *derive a specification from a given call site?*

RQ3 *Does* SPAIR *have the ability to generate specification patches?*

RQ4 *Can* SPAIR *generate patches for specifications with multiple bugs?*

Benchmarks: Our test subjects include 10 high-profile projects (i.e., high market capitalization) from EtherScan [80], whose source codes are available. The relevant statistics of these contracts are shown in Table 6.2. The prevalence of tokens in the benchmarks reflects that many of the real-world deployed Ethereum contracts are of this type. We note that many verified contracts hold a significant amount of funds (> 200M worth of USD) with millions of transactions. Most of the projects have over 250 lines of code and 20 functions. Each project is associated with a Solidity file, which typically contains multiple contracts including a main one as well as library or parent contracts. The specifications of standard functions (e.g., `transfer`, `transferFrom`) are derived from the prior works [95], [130]. For non-standard functions (e.g., `finalize`, `freeze`,

		Precondition		Postcondition	
Project	#Function	Time (s)	#Pre	Time (s)	#Post
BAT	12	2.00	5/0/7	0.65	0/10/2
BNB	13	2.80	7/0/6	0.74	0/12/1
HT	13	2.85	3/0/10	0.49	0/8/5
HOT	18	2.30	9/0/9	0.63	0/15/3
IOTX	22	3.57	10/0/12	0.92	0/19/3
QNT	14	2.35	4/0/10	0.59	0/12/ 2
MANA	19	2.23	11/0/8	0.64	0/16/3
ZIL	26	6.49	12/0/14	1.22	0/22/4
NXM	27	2.93	18/0/9	1.20	0/25/2
SHIB	23	2.45	5/0/18	0.82	0/21/2

TABLE 6.3: Results of SPAIR on inferring missing conditions. For each project, the table shows the number of functions (#Function), preconditions (#Pre), postconditions (#Post), and the execution time (Time). The column #Pre (or #Post) is formatted as X/Y/Z where X, Y, and Z represent the number of inferred conditions that are weaker, stronger, and identical to the original conditions respectively

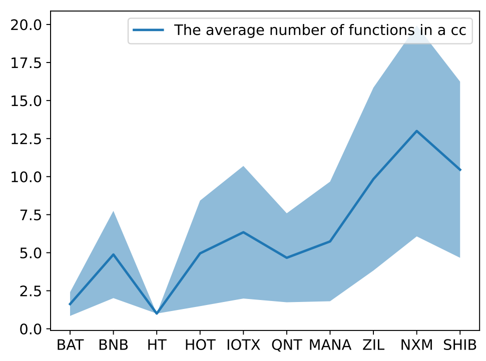
`unfreeze`), we develop specifications by considering the implementations. We systematically apply SPAIR to each contract and set timeout to be 5 minutes for each contract. All the experiments are conducted on a macOS 13.3 machine with 2.6 GHz 6-Core Intel Core i7 and 16GB of memory.

RQ1: Performance on missing specifications (P-patch, Q-patch) RQ1 aims to show that SPAIR can be used to recommend missing preconditions or postconditions. To answer this RQ, we systematically omit either precondition or postcondition of each correct specification in the test subjects and apply SPAIR to find the missing one. Every omitted condition has at most one suggested condition. The suggested condition is always verified against the implementation to guarantee its correctness. We ignore functions that have empty implementation. For example, given a function sub associated with the correct specification ensures($a \geq b$, r == a - b), we infer the missing precondition for ensures($\ldots$, r == a - b) and missing postcondition for ensures($a \geq b, \ldots$). The experimental results are reported in Table 6.3.

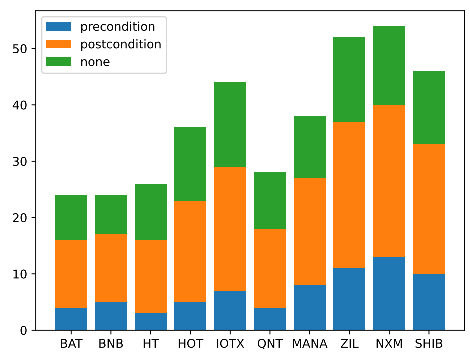
We can observe that SPAIR successfully infers a total of 374 missing conditions including 187 preconditions and 187 postconditions, and leaves none omitted conditions unaddressed. Among them, 84 inferred preconditions are weaker than the original ones and 160 inferred postconditions are stronger than the original ones. It takes a longer time for SPAIR to infer preconditions (3 seconds) compared to postconditions (0.79 seconds). The reason for this discrepancy is that inferring preconditions is more challenging than inferring postconditions. There are no stronger preconditions or weaker postconditions in the results. This confirms the correctness of our inference algorithm.

RQ2: Performance on call sites (F-patch) The primary focus of RQ2 is to demonstrate that SPAIR is able to recommend the specification of an invoked function. To answer this RQ, we enumerate all internal function calls and perform specification inference. Each internal function call is associated with at most one suggested specification. Since the suggested specification is inferred from the call site, there is a possibility that it may not be correct. The correctness is determined by verifying the suggested specification against the implementation. The experimental results are reported in Table 6.4.

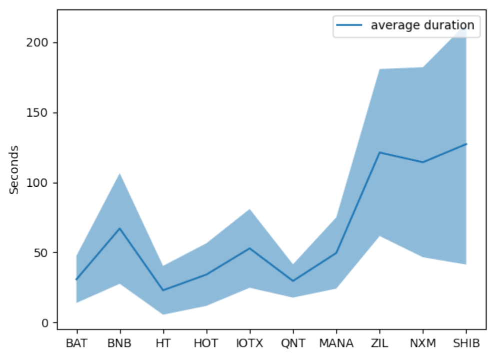
We can observe that SPAIR is able to infer specifications for 114 out of 117 internal calls, accounting for 97%. The failures on 3 internal calls are attributed to the following reasons. First, the presence of multiplication operators in the encoding adversely affects the inference process. Second, eliminating variables that are keys of mappings (e.g., eliminating `_from` from the expression `balances[_from]`) result in a timeout.



(A) Connected components



(B) Patch



(C) Duration

FIGURE 6.6: Results of patching specification for 10 projects

According to the experiment results, SPAIR takes the longest time, 10.85 seconds, to successfully generate 16 specifications for the internal calls in the ZIL project. In contrast, the NXM project, which has 20 internal calls, only needs 5.79 seconds. The notable difference is due to the substantial time spent waiting for a timeout (2 seconds) of two failed specifications in the ZIL project. A similar issue is also observed in the BAT project, where the execution time is 6.7 seconds. Since the majority of the internal calls are library calls (e.g., `sub`, `add`), we observe many identical post-conditions. By looking at the execution time and the number of functions (or internal functions) in Table 6.3 and Table 6.4, it is obvious that inferring specifications from the call sites is more expensive and less efficient compared to inferring preconditions or postconditions.

Project	#Internal	Time (s)	#Success	#Pre	#Post
BAT	3	6.70	2	0/2/0	0/0/1
BNB	11	7.64	11	0/11/0	0/0/11
HT	0	0.00	0	0/0/0	0/0/0
HOT	11	6.23	11	0/11/0	0/0/11
IOTX	12	3.67	12	0/12/0	5/0/7
QNT	8	3.52	8	0/8/0	0/0/8
MANA	12	4.80	12	0/11/1	4/0/8
ZIL	18	10.85	16	0/16/0	5/0/11
NXM	22	5.79	22	0/22/0	9/1/12
SHIB	20	3.62	20	0/20/0	14/0/6

TABLE 6.4: Results of SPAIR on inferring specifications from call sites. For each project, the table shows the number of internal calls (#Internal), successfully inferred specifications (#Success), preconditions (#Pre), post-conditions (#Post), and the execution time (Time).

RQ3: Performance on patching RQ3 aims to show that SPAIR is capable of suggesting patches when a single buggy specification is presented in the specification. This RQ evaluates Algorithm 7 entirely, while RQ1 and RQ2 compute P-patch (Definition 4), Q-patch (Definition 5), and F-patch (Definition 6) for specified functions. To answer this RQ, we systematically set default values to either precondition or post-condition of each correct specification in the test subjects and apply SPAIR to patch the incorrect one. That is, the default values of the precondition and post-condition are $true$ and $false$ respectively. It is noted that there are numerous cases where the specification remains correct even if the precondition is set to $true$, and in such cases, no patch needs to be generated. The experimental results are shown in Figure 6.6

In Algorithm 7, we group a set of related functions that have an impact on the patch into a connected component, and we address each connected component sequentially. The number of functions in each connected component reflects the complexity of a generated patch. It is because all functions in a connected component must be examined to generate a patch. Figure 6.6(a) shows the average number of functions that SPAIR attempts to patch, with the filled area representing the standard deviation. On average, across all projects, the connected component size is 7.39. Among the projects, HT has the smallest connected component size, while NXM has the largest connected component size. This discrepancy is due to the design differences between the two projects. In HT, each function is a leaf function that does not call any other functions. In contrast, NXM employs an advanced design pattern where a public function acts as an interface calling multiple private functions. Although the call depth in the NXM project does not exceed 5, its large connected component size is due to the presence of crucial functions like sub and div, which are used in many functions.

Figure 6.6(b) shows the number of top-ranked patches generated by SPAIR. The label *precondition* (#P) denotes the patch that patches the specification where its precondition is set to $true$. The label *postcondition* (#Q) represents the patch that patches the specification where its precondition is set to $false$. The label *none* (#N) indicates that the specification remains correct even if we set its precondition as $true$. We can observe that the total number of #N is 116, whereas the total number of #P is 70. This suggests that the majority of functions accept the precondition $true$ because the inputs have been sanitized by `require` statements. Additionally, the total number of #Q is 186, which is $2x$ the number of #P. This is because setting the postcondition to $false$ always leads to an incorrect specification.

Figure 6.6(c) shows the average time (in seconds) to generate a patch where the filled area represents the standard deviation. We can observe that the execution time is directly related to the size of the connected component. For patching a specification in HT, SPAIR requires approximately 22.86 seconds, whereas it takes around 127.25 seconds for a patch in SHIB. On average, SPAIR takes 64.92 seconds to generate a patch.

RQ4: Repairing specifications with multiple bugs RQ4 is designed to demonstrate that SPAIR is able to generate patches when there are multiple falsified specifications are presented. To answer this RQ, we measure the performance of SPAIR on more complicated test subjects. Recall that RQ3 sequentially modifies preconditions or postconditions by setting them to the default values and generating corresponding patches. Given a falsified specification, the correctness of other functions may be affected through call chains. Since all specifications are originally correct and thus, a single patch like P-patch, Q-patch, or F-patch can easily make all specifications become correct. This specific configuration could not demonstrate the robustness of our patching algorithm that patches incorrect specifications in a bottom-up manner. To address this limitation, we adopt an alternative configuration where we set the preconditions of all functions to $true$ and compute patches for each project. This setup greatly increases the complexity of test subjects because an accepted patch has to modify at most N functions in a project where N is the number of incorrect specifications. Additionally, SPAIR must follow the call chain in a bottom-up manner to produce patches. The experimental results are shown in Table 6.5.

SPAIR effectively generates patches for all 10 projects. In the MANA and ZIL projects, the generated patches modify the specifications of 8 functions, while only one specification is modified in the SHIB project. The discrepancy is because SHIB uses `require` statements to sanitize inputs, but other projects

Project	# Function	Time (s)	Depth
BAT	4	17.22	2
BNB	5	36.68	2
HT	3	15.69	1
HOT	5	24.97	2
IOTX	6	35.87	3
QNT	4	21.28	2
MANA	8	11.07	3
ZIL	8	80.08	5
NXM	3	45.85	4
SHIB	1	8.93	3

TABLE 6.5: Result of SPAIR on more complicated test subjects. For each project, the table shows the number of functions (#Function) such that their specifications are patched, the execution time (Time) in seconds, and the maximum call stack depth (Depth)

employ `assert` statements. The usage of `assert` statements leads to complicated preconditions. However, in practice, `assert` are `require` statements that exhibit nearly indistinguishable behavior.

From the results, it can be observed that the average call depth is 2.7. Among 10 projects, ZIL has the highest call depth 5. It is the main reason that SPAIR takes the longest time, 80.08 seconds, to complete. The strategy of SPAIR is to fix specifications in a bottom-up manner and a higher call depth results in a greater number of potential patches. Additionally, the execution time is also influenced by the number of incorrect specifications. For example, SPAIR only needs 8.93 seconds to patch the SHIB project, which has 1 incorrect specification. On average, SPAIR takes 29.76 seconds to produce a patch. Note that SPAIR implements a heuristic to minimize the number of patches. That is, SPAIR first generates a patch, computes the distance, and considers it as a pivot. During the generation of P-patch, Q-patch, or F-patch, the distance of an incomplete patch is computed, and if it exceeds the reference point, the incomplete patch is discarded. This heuristic eliminates the need to compute irrelevant patches and thus enhances the efficiency.

6.7 Related work

To the best of our knowledge, SPAIR is the first tool that is able patch specifications in a provably correct way.

SPAIR is closely related to many works on automated smart contract repair. We discuss the most relevant ones in the following. Elysium [34] and sGuard [45] combine template-based and semantic-based approaches to introduce patches. Elysium uses an outsourced bug localization and repairs smart contracts at bytecode level. While sGuard leverages its own bug localization and repairs contracts at the source-code level. EVMPatch [35] features a bytecode rewriting engine and leverages a template-based approach to patch integer overflow and access control bugs. SmartShield [33] extracts control and data dependencies from bytecode and source-code, then uses extracted semantics to fix insecure control flows and data operations. While there are many other program repair works, none of them focus on fixing specifications.

SPAIR is also related to smart contract verification approaches developed in the last few years. VeriSmart [66] relies on intra-procedure analysis to verify arithmetic safety. SmartACE [102] is a framework that runs multiple independent analyzers to verify user-annotated assertions. It transforms smart contracts into constrained Horn clauses for correctness verification. VerX [30] reduces the temporal properties of smart contracts to reachability problems and then applies state-of-the-art reachability checking.

SPAIR is broadly related to the many work on applying static analysis techniques on smart contracts. Securify [24] and Ethainter [103] are approaches that leverage a rewriting system (i.e., Datalog) to identify vulnerabilities through pattern matching. In terms of symbolic execution, Luu *et al.* presented the first engine to find potential security bugs in smart contracts [23]. Krupp and Rossow presented teEther [27] which finds vulnerabilities in smart contracts by focusing on financial transactions. Nikolic *et al.* presented MAIAN [28], which focuses on identifying trace-based vulnerabilities through a form of symbolic execution. Torres *et al.* presented Osiris [29] which focuses on discovering integer bugs.

SPAIR is remotely related to some works on invariant detection. Ernst et al. developed Daikon [97], a tool for dynamic detection of likely invariants. Daikon is capable of detecting invariants in C, C++, Java, and Perl programs. It defines a set of invariant templates and selects suitable invariants by analyzing them against data traces. The inferred invariants contain preconditions, post-conditions, and object invariants. Ye Liu et al. presented InvCon [131], an extension of Daikon designed to automatically detect contract invariants from transaction histories. It can be used to mitigate the absence of specifications for Ethereum smart contracts. SPAIR is not exclusive to smart contracts. It can be extended to infer specifications for programs implemented in other languages.

Finally, SPAIR is remotely related to approaches on testing smart contracts. ContractFuzzer [26] is a fuzzing engine that checks 7 different types of vulnerabilities. sFuzz [43] is another fuzzer that extends ContractFuzzer by using feedback from test case execution to generate new test cases.

6.8 Conclusion

In this work, we have proposed a novel approach to identify and patch buggy specifications of smart contracts. Our approach automatically fixes incorrect specifications in a bottom-up manner with provable correctness. We have implemented our approach in a tool and thoroughly evaluated the tool on 10 high-profile smart contracts. The experiment results show the usefulness of our approach, i.e., SPAIR is capable of inferring missing correct specifications, and patching specifications correctly while introducing a minor difference from original smart contracts. In the future, we intend to improve the performance of SPAIR further on complicated smart contracts.

Chapter 7

Conclusion and Future Work

In this chapter, we summarize our work and discuss future direction.

7.1 Conclusion

Blockchain is a public database that stores data in blocks linked together via cryptographic hashes. Once data is written to the blockchain, it is irreversible. Blockchain has the potential to revolutionize many industries, especially with the power of smart contracts. However, the flawed design, as well as, the complexity of deployed smart contracts make them error-prone. Many security incidents with substantial financial losses raise serious concerns about the security problem of smart contracts. Although many relevant research works have been conducted to mitigate the problems, they left many unsolved issues. In this thesis, we ensure the safety and correctness of smart contracts by proposing methods to automatically detect and eliminate vulnerabilities in smart contracts. We presented our works in two separate parts: *smart contract vulnerability detection* and *smart contract repair* where each part consists of two works.

In the first part, we rely on fuzzing to detect generic vulnerabilities and compositional verification to detect specification violations. In Chapter 3, we presented sFuzz, an adaptive fuzzing engine for smart contracts. It combines the strategy in the AFL [37] and an efficient lightweight multi-objective adaptive algorithm to improve the quality of the test suite. That is, sFuzz relies on coverage-guided fuzzing strategies of AFL to generate test inputs and execute them against the main contract to record the execution behaviors such as a reentrancy attack from the attacker contract to the main contract. It keeps one best seed (i.e., the seed with the smallest branch distance) for each just-missed branch. This adds a small number of test inputs to the original test suite, but it increases the chance of covering new branches. Subsequently, in Chapter 4, we presented iContract, a compositional verification engine that verifies the correctness of smart contracts. It allows users to specify functional level properties in the form of pre/post-conditions and verify them compositionally. As we heavily rely on manually drafted specifications, iContract supports traditional invariants such as loop invariants, and contract invariants. It further supports writing contract invariants and revert specifications, which describe the behaviors of a function call and reverted execution flow respectively.

In the second part, we take different angles to repair contracts. We repair the smart contract implementation and its specifications respectively. In Chapter 5, we introduced an algorithm that repairs the implementation at the source code level but identifies vulnerabilities at the bytecode level. sGuard locates vulnerabilities by leveraging a tainting algorithm that analyzes control and data dependency in stack, memory, and storage. Each vulnerability is defined in the form of dependencies and is patched using the corresponding templates. The vulnerability definition is over-approximated to ensure that our repaired code is always safe. It is noted that sGuard is able to prove the correctness of the generated patch. Subsequently, in Chapter 6, we presented SPAIR, a tool that relies on abductive inference and constraint solving to recommend correct and minimal specifications. SPAIR relies on a compositional verification engine to identify buggy specifications and generate patches for them. All generated patches are ranked and only the best one is selected as the final result. It is noted that SPAIR is the first tool that is able to patch specifications in a provably correct way.

In summary, this thesis provides multiple approaches to ensure the safety and correctness of smart contracts. Given a contract, sFuzz fuzz it to detect vulnerabilities, iContract verifies it against its specifications. While sGuard patches vulnerable code, SPAIR patches buggy specifications. Although our work is undoubtedly useful, it may not be able to protect smart contracts from new vulnerabilities and attack methods. Our results shed some light on smart contract security and provide useful resources and background for future research projects.

7.2 Future Work

Smart contracts are rapidly developing and so are new ways of attacking smart contracts. One of our immediate future work is to extend our approaches to handle multiple emerging vulnerabilities such as *Maximum Extract Value* (MEV) attacks [132]–[134].

MEV refers to the situation where attackers attempt to include, exclude, or change the order of transactions in a block to gain value that exceeds the standard block reward and gas fees. There are a few common types of MEVs as follows.

DEX arbitrage: This refers to the situation where two DEXes sell a token at two different prices. An attacker buys and sells the token on a lower-price DEX and a higher-price DEX respectively.

Liquidations: This is a well-known MEV opportunity in lending protocols. Lending protocols allow anyone to liquidate the collateral of a borrower to receive a hefty liquidation fee if the borrowing amount exceeds the allowed (e.g., 30% of the collateral). Because of this, an attacker can listen to the transaction pool and determine which borrowers can be liquidated.

Sandwich trading: This refers to the situation where an attacker font-run a large DEX trade to buy tokens at a lower price and sell tokens after the large DEX trade at a higher price.

Another line of future work that we would like to pursue is the automatic synthesis of smart contracts implementation with the help of large language models (LLM). Implementing a smart contract from scratch is undeniably risky. We would like to develop a tool that relies on trusted libraries such as Openzeppelin [106] to minimize the use of unsafe code. For example, we first synthesize a contract that satisfies the given specifications. Then, we instruct a LLM to enhance its safety such as replacing the generated code with equivalent libraries, adding a modifier `nonReentrant` to functions to avoid re-entrancy attack, or adding interface `Pausable` for emergency stop capability.

Bibliography

[1] J. Frankenfield, “Understanding double-spending and how to prevent attacks,” *Investopedia*, 2023.

[2] A. Hankin, “Bitcoin pizza day: Celebrating the $250 million pizza order,” *Investopedia*, 2023.

[3] CoinMarketCap, *Top 100 crypto coins by market capitalization*, `https://coinmarketcap.com/coins/`, 2023.

[4] J. Frankenfield, “Cryptographic hash functions: Definition and examples,” *Investopedia*, 2023.

[5] A. Lu, *Opcodes for the evm*, `https://ethereum.org/en/developers/docs/evm/opcodes/`, 2023.

[6] D. Muhs, *Reentrancy*, `https://swcregistry.io/docs/SWC-107/`, 2023.

[7] T. S. Team, *Solidity*, `https://soliditylang.org/`, 2023.

[8] T. V. Team, *Vyper documentation*, `https://docs.vyperlang.org/en/stable/`, 2023.

[9] Y. Hirai, *Bamboo: A language for morphing smart contracts*, `https://github.com/pirapira/bamboo`, 2023.

[10] CoinMarketCap, *Ethereum Price Today*, `https://coinmarketcap.com/currencies/ethereum/`, 2023.

[11] Blockchains, *Decentralized autonomous organization (dao) framework*, https://github.com/blockchainsllc/DAO, 2023.

[12] E. Foundation, *The history of ethereum*, https://ethereum.org/en/history/, 2023.

[13] V. Buterin, *Hard fork completed*, https://blog.ethereum.org/2016/07/20/hard-fork-completed, 2016.

[14] M. E. Peck, *"Hard Fork" Coming to Restore Ethereum Funds to Investors of Hacked DAO*, https://spectrum.ieee.org/hacked-blockchain-fund-the-dao-chooses-a-hard-fork-to-redistribute-funds, 2016.

[15] Etherscan, *Ethereum transaction hash (txhash) details*, https://tinyurl.com/3h2k8uyx, 2017.

[16] Etherscan, *Ethereum transaction hash (txhash) details*, https://tinyurl.com/4fbetkkv, 2017.

[17] Etherscan, *Ethereum transaction hash (txhash) details*, https://tinyurl.com/nh9wpts, 2017.

[18] Etherscan, *Ethereum transaction hash (txhash) details*, https://tinyurl.com/mn8bsczx, 2017.

[19] devops199, *Anyone can kill your contract*, https://github.com/openethereum/parity-ethereum/issues/6995, 2017.

[20] DeFiHackLabs, *Defi hacks reproduce - foundry*, https://github.com/SunWeb3Sec/DeFiHackLabs, 2023.

[21] Rekt, *Rekt Leaderboard*, https://rekt.news/leaderboard/, 2023.

[22] SlowMist, *Slowmist hacked*, https://hacked.slowmist.io/, 2023.

[23] 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*, 2016, pp. 254–269.

[24] 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*, 2018, pp. 67–82.

[25] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[26] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[27] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium*, 2018, pp. 1317–1333.

[28] 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*, 2018, pp. 653–663.

[29] C. F. Torres, J. Schütte, *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[30] 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*, 2020, pp. 1661–1677.

[31] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig, "SmartPulse: Automated checking of temporal properties in smart contracts," in *2021 IEEE Symposium on Security and Privacy*, 2021, pp. 555–571.

[32] Á. Hajdu and D. Jovanović, "Solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments: 11th International Conference*, 2020, pp. 161–179.

[33] 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*, 2020, pp. 23–34.

[34] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 115–128.

[35] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and automated patching of ethereum smart contracts," in *30th USENIX Security Symposium*, 2021, pp. 1289–1306.

[36] T. D. Nguyen, *Toolkit*, https://github.com/duytai/toolkit.

[37] M. Zalewski, *Technical "whitepaper" for afl-fuzz*, http://lcamtuf.coredump.cx/afl/technical_details.txt, 2019.

[38] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, pp. 226–247, 2010.

[39] P. McMinn, "Search-based software test data generation: A survey," *Software testing, Verification and reliability*, pp. 105–156, 2004.

[40] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[41] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering*, 2013, pp. 802–811.

[42] A. Marginean, J. Bader, S. Chandra, *et al.*, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 269–278.

[43] 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 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[44] T. D. Nguyen, L. H. Pham, J. Sun, and Q. L. Le, "An idealist's approach for smart contract correctness," in *24nd International Conference on Formal Engineering Methods*, 2023.

[45] T. D. Nguyen, L. H. Pham, and J. Sun, "sGuard: Towards fixing vulnerable smart contracts automatically," in *2021 IEEE Symposium on Security and Privacy*, 2021, pp. 1215–1229.

[46] T. D. Nguyen, L. H. Pham, J. Sun, Y. Wan, and F. Song, "SPAIR: Towards repairing smart contract specification," in *Arxiv*, 2023.

[47] Tether, *Tether token*, https://tether.to/en/, 2023.

[48] BNB, *What is bnb?* https://www.binance.com/en/bnb, 2023.

[49] AAVE, *Aave liquidity protocol*, https://aave.com/, 2023.

[50] Compound, *Compound*, https://compound.finance/, 2023.

[51] Uniswap, *Uniswap protocol*, https://uniswap.org/, 2023.

[52] SushiSwap, *Sushiswap protocol*, https://www.sushi.com/, 2023.

[53] V. Buterin, *Ethereum whitepaper*, https://ethereum.org/el/whitepaper/, 2023.

[54] Joshua, *Proof-of-work (PoW)*, https://ethereum.org/fil/developers/docs/consensus-mechanisms/pow/, 2023.

[55] M. Eth, *Live Ethereum TPS data*, https://ethtps.info/, 2023.

[56] S. Foundation, *Web3 Infrastructure for Everyone*, https://solana.com/, 2023.

[57] CryptoKitties, *Collect and breed digital cats!* https://www.cryptokitties.co/, 2023.

[58] L. research team, *Arbitrum One*, https://l2beat.com/scaling/projects/arbitrum, 2023.

[59] L. research team, *ZkSync Era*, https://l2beat.com/scaling/projects/zksync-era, 2023.

[60] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, 1997.

[61] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[62] T. S. Team, *Solidity 0.8.21 documentation*, https://docs.soliditylang.org/en/v0.8.21/, 2023.

[63] C. Diligence, *Ethereum Smart Contract Best Practices*, https://consensys.github.io/smart-contract-best-practices/, 2023.

[64] M. Rodler, W. Li, G. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[65] OpenZeppelin, *OpenZeppelin Docs*, https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard, 2023.

[66] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy*, 2020, pp. 1678–1694.

[67] *September Hacks*, https://www.insurace.io/blog/?p=3519, 2023.

[68] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," *arXiv*, 2016.

[69] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, 2017.

[70] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, pp. 2–1, 2014.

[71] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[72] B. Marino and A. Juels, "Setting standards for altering and undoing smart contracts," in *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, 2016, pp. 151–166.

[73] P. Daian, *Analysis of the DAO exploit*, https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2023.

[74] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 291–302.

[75] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018, pp. 745–761.

[76] E. Foundation, *Aleth: Ethereum C++ client, tools and libraries*, https://github.com/ethereum/aleth/, 2023.

[77] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[78] J. H. Holland, "Genetic algorithms," *Scientific american*, pp. 66–73, 1992.

[79] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, pp. 122–158, 2018.

[80] Etherscan, *Etherscan*, https://etherscan.io/, 2023.

[81] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[82] 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*, 2020, pp. 557–560.

[83] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium*, 2018.

[84] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016, pp. 91–96.

[85] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. 2002.

[86] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*, 2016, pp. 79–94.

[87] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust: 6th International Conference*, 2017, pp. 164–186.

[88] M. Fröwis and R. Böhme, "In code we trust?" In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2017, pp. 357–372.

[89] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 442–446.

[90] S. Palladino, "The parity wallet hack explained," *OpenZeppelin blog*, 2017.

[91] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, p. 54, 2018.

[92] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy*, 2021, pp. 103–119.

[93] 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*, 2020, pp. 438–453.

[94] SmartContractSecurity, *Smart Contract Weakness Classification*, https://swcregistry.io/, 2023.

[95] F. Vogelsteller and V. Buterin, "Erc-20: Token standard," *Ethereum Improvement Proposals*, 2015.

[96] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, "Demystifying loops in smart contracts," in *35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 262–274.

[97] M. D. Ernst, J. H. Perkins, P. J. Guo, *et al.*, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, pp. 35–45, 2007.

[98] P. W. O'Hearn, "Incorrectness logic," *Proceedings of the ACM on Programming Languages*, pp. 1–32, 2019.

[99] E. Foundation, *Natspec format*, https://docs.soliditylang.org/en/v0.8.17/natspec-format.html, 2023.

[100] T. D. Nguyen, *Dataset*, https://anonymous.4open.science/r/zero1-0DEE/, 2023.

[101] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and E. Zhong, “Fast and reliable formal verification of smart contracts with the move prover,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2022, pp. 183–200.

[102] S. Wesley, M. Christakis, J. A. Navas, R. Trefler, V. Wüstholz, and A. Gurfinkel, “Verifying solidity smart contracts via communication abstraction in smartace,” in *Verification, Model Checking, and Abstract Interpretation: 23rd International Conference*, 2022, pp. 425–449.

[103] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, “Ethainter: A smart contract security analyzer for composite vulnerabilities,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[104] P. Technologies, “A postmortem on the parity multi-sig library self-destruct,” *Parity Blog*, 2017.

[105] V. Buterin, “Thinking About Smart Contract Security,” *Ethereum Blog*, 2016.

[106] OpenZeppelin. “Openzeppelin contracts is a library for secure smart contract development.” (2023).

[107] J. Jiao, S. Kan, S. Lin, D. Sanan, Y. Liu, and J. Sun, “Semantic understanding of smart contracts: Executable operational semantics of solidity,” in *2020 IEEE Symposium on Security and Privacy*, May 2020, pp. 1695–1712.

[108] Consensys, *Known Attacks*, `https://consensys.github.io/smart-contract-best-practices/attacks/`, 2023.

[109] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "Scompile: Critical path identification and analysis for smart contracts," in *International Conference on Formal Engineering Methods*, Springer, 2019, pp. 286–304.

[110] M. Mossberg, F. Manzano, E. Hennenfent, *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.

[111] C. N. Fischer, *A worklist algorithm for dominators*, http://pages.cs.wisc.edu/~fischer/cs701.f08/lectures/Lecture19.4up.pdf, 2020.

[112] Crytic, *Manage and switch between Solidity compiler versions*, https://github.com/crytic/solc-select, 2020.

[113] V. Buterin, "EIP-170: Contract code size limit," *Ethereum Improvement Proposals*, 2016.

[114] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, pp. 1–40, 2009.

[115] J. H. Perkins, S. Kim, S. Larsen, *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[116] Y. Hirai, "Formal verification of deed contract in ethereum name service," 2016.

[117] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, p. 9, 2008.

[118] S. K. Lahiri and S. Qadeer, "Call invariants," in *NASA Formal Methods Symposium*, 2011, pp. 237–251.

[119] I. Dillig and T. Dillig, "Explain: A tool for performing abductive inference," in *Computer Aided Verification*, 2013, pp. 684–689.

[120] S. Wu, D. Wang, J. He, *et al.*, "Defiranger: Detecting price manipulation attacks on defi applications," *arXiv*, 2021.

[121] C. Diligence, "Oracle Manipulation," *Ethereum Smart Contract Best Practices*, 2023.

[122] BlockSec, "Public transfer vulnerability of the Tether Gold smart contract," *Blocksec Medium Blog*, 2023.

[123] 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*, pp. 1–27, 2018.

[124] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[125] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, pp. 184–195, 1960.

[126] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Tech. Rep., 2008.

[127] 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*, 2019, pp. 8–15.

[128] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[129] Z. Liao, S. Song, H. Zhu, *et al.*, "Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts," *IEEE Transactions on Software Engineering*, pp. 777–801, 2022.

[130] Z. V. Zhou, "Token minting and burning," *Ethereum Improvement Proposals*, 2022.

[131] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for ethereum smart contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–4.

[132] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *2021 IEEE Symposium on Security and Privacy*, 2021, pp. 919–936.

[133] L. Zhou, X. Xiong, J. Ernstberger, *et al.*, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy*, 2023, pp. 2444–2461.

[134] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" In *2022 IEEE Symposium on Security and Privacy*, 2022, pp. 198–214.