Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

9-2020

Smart contract repair

Xiao Liang YU
National University of Singapore

Omar AL-BATAINEH
National University of Singapore

David LO
Singapore Management University, davidlo@smu.edu.sg

Abhik ROYCHOUDHURY
National University of Singapore

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Software Engineering Commons

Citation

YU, Xiao Liang; AL-BATAINEH, Omar; LO, David; and ROYCHOUDHURY, Abhik. Smart contract repair. (2020). *ACM Transactions on Software Engineering and Methodology*. 29, (4), 1-32.
Available at: https://ink.library.smu.edu.sg/sis_research/5623

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

arXiv:1912.05823v3 [cs.SE] 20 May 2020

Smart Contract Repair

XIAO LIANG YU, National University of Singapore, Singapore
OMAR AL-BATAINEH, National University of Singapore, Singapore
DAVID LO, Singapore Management University, Singapore
ABHIK ROYCHOUDHURY*, National University of Singapore, Singapore

Smart contracts are automated or self-enforcing contracts that can be used to exchange assets without having to place trust in third parties. Many commercial transactions use smart contracts due to their potential benefits in terms of secure peer-to-peer transactions independent of external parties. Experience shows that many commonly used smart contracts are vulnerable to serious malicious attacks which may enable attackers to steal valuable assets of involving parties. There is therefore a need to apply analysis and automated repair techniques to detect and repair bugs in smart contracts before being deployed. In this work, we present the first general-purpose automated smart contract repair approach that is also gas-aware. Our repair method is search-based and searches among mutations of the buggy contract. Our method also considers the gas usage of the candidate patches by leveraging our novel notion of *gas dominance relationship*. We have made our smart contract repair tool SCRepair available open-source, for investigation by the wider community.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Security and privacy →** *Software security engineering*.

ACM Reference Format:
Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart Contract Repair. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (May 2020), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts are automated or self-enforcing programs which currently underpin many online commercial transactions. A smart contract is a series of instructions or operations written in special programming languages which get executed when certain conditions are met. Typically, smart contracts are running on the top of blockchain systems, which are distributed systems whose storage is represented as a sequence of blocks. The key attractive property of smart contracts is mainly related to their ability to eliminate the need of trusted third parties in multiparty interactions, enabling parties to engage in secure peer-to-peer transactions without having to place trust in external parties (i.e., outside parties which help to fulfill the contractual obligations).

While smart contracts are commonly used for commercial transactions, many malicious attacks in the past were made possible due to poorly written or vulnerable smart contracts. The code executed by smart contracts can be complex. There is therefore a need for testing (e.g. [16, 24]), analysis (e.g. [18]) and verification (e.g. [36]) of smart contracts. In this paper, we take the technology

*Corresponding Author

Authors' addresses: Xiao Liang Yu, National University of Singapore, Singapore, xiaoly@comp.nus.edu.sg; Omar Al-Bataineh, National University of Singapore, Singapore, omerdep@yahoo.com; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@comp.nus.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1049-331X/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for enhancing reliability of contracts one step further: once vulnerabilities in smart contracts are detected, we seek to automatically *repair* the vulnerabilities.

Automated program repair [11] is an emerging technology for automatically fixing errors and vulnerabilities in programs via search, symbolic analysis, program synthesis and learning. The successful application of automated repair techniques to traditional programs [20, 21, 23, 26–29, 40] raises the question of whether these techniques can be also applied to fix bugs in smart contracts. Several different approaches have been developed to automatically repair bugs in traditional programs which can be classified mainly into two categories: heuristic repair approaches [20, 21, 26] and constraint-based repair approaches [23, 27–29, 40]. The inputs to these approaches are a buggy program and a correctness criterion (often given as a test suite). The automated repair approaches return a (often minimal) transformation of the buggy program, so that the transformed program passes all the tests in the given test-suite.

In practice, the implications of unfixed bugs in smart contracts can be more serious than the typical non-security sensitive programs for several reasons. First, smart contracts are open for inspection and running on a decentralized network, the whole program state of smart contracts is transparent to everyone. Second, the generated patch for a vulnerable smart contract should not only fix the detected vulnerabilities but also needs to be mindful of the gas consumption of the resultant patched program. The blockchain system on which the contract will be running typically has a gas usage limit. Third, the quality of the generated patch for a vulnerable smart contract is a major design issue to be considered as smart contracts are typically used for commercial transactions. In fact, malicious agents may take advantage of unfixed bugs in smart contracts to steal some valuable assets of the parties involved.

In this work, we develop an automated smart contract repair algorithm using genetic programming search. Given a vulnerable smart contract and test suite, we conduct a parallel, biased random search for a set of edits to the contract that fixes a given vulnerability without breaking any test that previously passed. The bias in the search comes from the objective function driving the search. The parallelization strategy consists of splitting the search space into mutually-exclusive (disjoint) sub-spaces, where patches in each sub-space are concurrently and independently generated and validated. We introduce also the notion of gas dominance level for smart contracts which enables us to compare the quality of patches based on their runtime gas. The gas dominance level can be used to compare the quality of generated patches. This also emphasizes our position is that automated repair of smart contracts needs to be gas-aware.

To evaluate the effectiveness of our genetic repair algorithm, we constructed a dataset of vulnerable smart contracts taken from the Ethereum mainnet network, which is the main network wherein actual transactions of smart contracts take place on a distributed ledger. Hence, our constructed dataset consists of real-world smart contracts. During our evaluation, we considered 20 vulnerable contracts which have been selected randomly from the constructed dataset while taking into consideration the class of detected vulnerabilities and the complexity of the vulnerable contracts. The vulnerable contracts have been selected in a way such that most of the common classes of vulnerabilities that are typically made by smart contract developers are covered when evaluating the genetic algorithm. However, to understand and draw some valid conclusions about the factors affecting the correctness and quality of patches generated by the algorithm, we have evaluated the algorithm under many different settings and configurations. Examples of such settings include: (i) enabling/disabling the gas calculation of generated patches, (ii) varying the size of time budget allocated to the algorithm. Our genetic algorithm was able to fully repair 10 vulnerable smart contracts from the selected set of 20 vulnerable contracts, achieving a 50% success rate. It is interesting to mention that most of the selected vulnerable contracts have multiple bugs and we therefore assert a vulnerable contract as repaired if all detected bugs are repaired.

Contributions. We summarize our main contributions as follows.

- We present the first automated smart contract repair approach that is gas-optimized and vulnerability-agnostic. The approach is inspired by genetic programming and can be used to generate a patch for a given vulnerable smart contract.
- We describe a parallel genetic repair algorithm that can be used to split the large search space of candidate patches into smaller mutually-exclusive search spaces which can be processed independently. The presented parallel algorithm helps to process large number of candidate patches in a short computational time and therefore, in contrast to previous repair approaches, repairs can be generated faster. It also improves the scalability of genetic repair algorithms so that large real-world contracts can be repaired.
- We show how to integrate gas-awareness into the repair of smart contracts. This is crucial for smart contracts as excessive unnecessary gas consumption of contracts can lead to financial loss or out-of-gas exceptions when running the contract on a public blockchain network. It is therefore necessary to reduce the cost of running the contract and also the possibility of introducing new out-of-gas exceptions when repairing a vulnerable smart contract. We introduce a simple yet effective gas ranking approach with the novel notion of *Gas Dominance Level* that can be used to rank generated patches of a given vulnerable smart contract during the patch generation. In general, the gas consumption of a given smart contract can be a non-constant bound which can be described as a parametric gas formula that takes into consideration both static and dynamic parameters that affect the cost of the contract including the instruction gas, memory gas, stack gas, and storage gas. We provide an acceleration technique to quickly compare candidate patches in terms of gas consumption, by introducing the concept of *gas dominance levels*.
- Based on the above described techniques, we develop a fully automated repairing tool for smart contracts (which we call SCRepair) which is integrated with a gas ranking approach to generate an gas-optimized secure contract. Our tool can both detect and repair security vulnerabilities in smart contracts. It is does so by integrating the tool SCRepair with the powerful smart contract security analyzer Oyente [24] and Slither [8]. We demonstrate that our approach is effective in fixing bugs for real-world smart contracts. Our approach can deal with bugs whose fixes involve multi-line changes. Our smart contract repair tool and dataset is publicly available in GitHub from https://SCRepair-APR.github.io

2 BLOCKCHAIN AND SMART CONTRACTS

The blockchain technology is a distributed database that maintains records and transactions in a decentralized fashion. It has been adopted in many applications to increase security and reliability and to avoid the need for a trusted third party. The transactions on blockchain are available to all the parties in the network in real-time and all the parties are allowed to interact with each other in a distributed manner. It uses state-of-the-art cryptography, and hence it enables parties to engage in secure peer-to-peer transactions. The decentralized nature of blockchain makes it suitable for many applications including decentralized cloud storage with provenance[42], general health care management, IoT data sharing with assured integrity[22], and general commercial transactions.

Smart contracts are one of the most successful applications of the blockchain technology. They currently underpin many online commercial transactions which are typically running on the top of blockchain systems. A smart contract is a special computer program whose executions are done in a decentralized and tamper-proof manner. The key attractive property of smart contracts is mainly related to their ability to eliminate the need of trusted third parties in multiparty interactions.

Smart contracts allow for decentralized automation by facilitating, verifying, and enforcing the conditions of an underlying agreement.

Ethereum is the most popular blockchain platform supporting smart contracts. It supports a feature called Turing-completeness that allows the creation of practically useful smart contracts. Smart contracts are typically written using the programming language "Solidity". Note that everything executed on Ethereum costs some gas for giving the miners incentive to perform the computations [38]. For example, executing an ADD instruction costs 3 units of gas. Storing a byte costs 4 units or 68 units of gas, depending on the value of the byte (zero or non-zero). Hence, any slight mutation to the source code of a smart contract can change the gas usage of the contract tremendously (and hence the amount of money that the parties of a transaction need to pay when running the smart contract on a real blockchain network).

To develop a better understanding of blockchain and smart contracts, let us consider an example. Suppose that Bob would like to sell a property (house) to Alice and Alice is willing to pay 100 Ether (a cryptocurrency) as a price for that property and that Bob is happy with Alice's offer. After some discussion, they agreed to proceed with their business transaction and wish to perform it in an automated way by taking advantage of blockchain and smart contracts. From the given description of the problem, one can see that there are three main conditions that any possible solution to the problem needs to satisfy: (1) Bob has legal ownership of the property that he is selling (2) Alice can get the ownership of Bob's property only if she transferred 100 Ether to Bob, and (3) Bob can get 100 Ether from Alice only if he transferred the ownership of his property to Alice. The transaction can be said to be successful if upon completion, the ownership of Bob's property is transferred to Alice while Bob receives 100 Ether.

Suppose that Alice and Bob perform their transaction using the smart contract given in Fig. 1 written in the most popular smart contract programming language Solidity. The code consists of a number of functions needed in order to perform the commercial transaction in an atomic way. The function `transferA` is used by Alice to transfer 100 Ether to the smart contract and hence when this function is executed, the money comes under the control of the smart contract. The function `transferB` is used by Bob to inform the smart contract that the ownership of his property has been transferred to the smart contract. So that after executing the functions `transferA` and `transferB`, the smart contract is supposed to hold both the money of Alice and the property of Bob. The function `finalize` is used to finalize the transaction by transferring the money from Alice to Bob and the property from Bob to Alice. The smart contract provides also two more functions, namely `abortA` and `abortB` which are available to both Alice and Bob respectively. The goal of these functions is to protect the parties from the situation where the purchase is canceled halfway while the smart contract has already held the assets from the parties, so that the parties can get back their assets from this contract.

Recently, there has been a growing interest in verification and validation of smart contracts [1, 14, 16, 18, 37], as vulnerabilities in smart contracts can have serious adverse consequences. Therefore, a number of vulnerability detection tools have been developed for smart contracts including Oyente [24], Slither[8], and ContractFuzzer [16]. In general, smart contract vulnerabilities can be categorized into three categories [3]: (i) vulnerabilities at the blockchain level, (ii) vulnerabilities at the Ethereum virtual machine level, and (iii) vulnerabilities at the source code level. In this work, we are interested on the vulnerabilities that can be repaired at the level of source code.

Based on our conducted literature review on recent research work on smart contracts [3, 4, 6, 7, 12, 16, 24, 35, 36], we summarize in Table 1 some selected popular vulnerabilities that can be detected using the tools Oyente [24] and Slither [8]. Table 1 shows a summary of these widely studied vulnerabilities. We give a detailed description of these classes of vulnerabilities in section 6.

```
contract CommercialTransaction {
    bool transferredA = false; bool transferredB = false;

    function transferA() public payable {
        if(msg.sender == Alice && msg.value == 100 ether) {
            transferredA = true;
        }
    }
    function transferB() public {
        // Bob should transfer the house ownership to this
        // contract before calling this function
        if(msg.sender == Bob && hasHouseOwnership(address(this))) {
            transferredB = true;
        }
    }
    function finalize() public {
        if(transferredA && transferredB) {
            transferredA = false; transferredB = false;
            transferHouseOwnership(Alice); Bob.transfer(100 ether);
        }
    }
    function abortA() public {
        if(msg.sender == Alice && transferredA) {
            transferredA = false; Alice.transfer(100 ether);
        }
    }
    function abortB() public {
        if(msg.sender == Bob && transferredB) {
            transferredB = false; transferHouseOwnership(Bob);
        }
    }
}
```

Fig. 1. A smart contract written in Solidity language that allows two parties to be involved in a commercial transaction to sell some property. `msg.sender` represents the party calling the function. Certain definitions are omitted for brevity.

Table 1. Selected smart contract vulnerabilities that can be fixed by modifying Solidity source code

Class of vulnerability	**References**
Exception disorders / Mishandled exceptions / Gasless send	[3, 4, 7, 16, 24, 35, 36]
Reentrancy	[3, 4, 7, 16, 24, 35, 36]
Integer overflow / Integer underflow / Unchecked math	[7, 24, 35, 36]
Transaction order dependence / Unpredictable state	[3, 7, 24, 36]

3 THE SMART CONTRACT REPAIR PROBLEM

Recent advances in program repair techniques [11] have raised the possibility of developing program repair technology for smart contracts. In this section, we discuss the automated smart contract repair problem together with the set of challenges that might be encountered in repairing smart

contracts. We also discuss the key differences between the smart contract repair problem and the traditional program repair problem.

Problem 1. *(**Automated smart contract repair problem**). Consider a vulnerable smart contract C with a set of detected vulnerabilities U, a test suite T and a maximum gas usage bound L, the automated smart contract repair problem is the problem of developing an algorithm that takes as input (C, U, T, L) and produces as an output a new contract C' that is similar to C but has all vulnerabilities in U fixed, passing all tests in T, and the maximum gas usage of feasible execution paths should be less than or equal to L.*

The smart contract repair problem is similar to the traditional program repair problem. However, the smart contract repair problem introduces some extra computational complexity as the patch generation needs to be gas-aware. It is also highly desirable for the patches to signify readable and small changes, so that the patched contract is easily comprehensible. Overall, we would want the the (syntactic) structure of the vulnerable contract to be maximally preserved.

Since detailed formal specifications of intended program behavior are typically unavailable, program repair uses weak correctness criteria, such as an assertion of existence of vulnerabilities by vulnerability detector and a test suite. Therefore, the validity of patches is relative to the chosen vulnerability detector and the available test cases.

As mentioned earlier, the generated patches for smart contracts need to meet more criteria than those generated for traditional programs. This is mainly due to the fact that smart contracts are typically running on the top of the blockchain systems, which impose certain constraints on the total computational resources used by the contract. The execution of the smart contract needs to comply with the gas usage constraints imposed by the blockchain system. Note that if the running smart contract exceeds the allowed upper bound limit of the gas usage, the execution of the contract will be interrupted and a "out-of-gas" exception will be thrown.

Definition 1. *(**Validity criteria of generated patches**). Given a vulnerable smart contract C with a set of detected vulnerabilities U and a test suite T that consists of two sets: the failing tests T_F and the passing tests T_P. Suppose that the contract C is running on the top of a blockchain system B and that the maximum allowed gas usage available to the contract is bounded by L. We say that the new patched smart contract C' is a valid plausibly fixed contract if it satisfies the following requirements.*

(1) The contract C' is not vulnerable to the vulnerabilities in U.
(2) The contract C' passes all tests in T_F.
(3) The contract C' does not break any test in T_P.
(4) There is no feasible execution path in C' whose total gas consumption exceeds the bound L.

Typically, the bound L imposed on the gas usage of the contract is determined by the involving parties of the transaction, the structure and semantics of the smart contract, and the block gas limit of the blockchain. Such bound (if known) can be incorporated in the patch generation process for vulnerable contracts in order to avoid introducing new out-of-gas exceptions. Note that requirement 4 of Definition 1 can be checked by enumerating all feasible paths in the patched contract C' and then verifying that there is no feasible path that exceeds the bound L.

In addition to the above correctness requirements, we are also interested in certain desirable properties indicating patch quality, as described in the following.

(1) **The simplicity of the patch**. The simplicity of the edited contract can be measured in terms of the number of edits that have been made to the original contract.
(2) **The cost of the patch**. The cost of the contract can be measured in different ways. We choose here the average gas usage as a metric to measure the cost of the contract.

To evaluate the quality requirements of a generated patch we introduce two functions, namely $diff(C, C^{'})$ and $cost(C^{'})$. The function $diff(C, C^{'})$ returns a numerical value that specifies how much the edited contract $C^{'}$ differs from the original vulnerable contract C. Replacing expressions, inserting of new statements, and moving/deleting of statements will be counted when computing $diff(C, C^{'})$. Overall, $diff(C, C')$ captures the edit distance between two smart contracts C and C'.

The function $cost(C)$ computes the average cost of gas usage of a given smart contract. Recall that every single operation that takes part in the blockchain network consumes some amount of gas. Gas is what is used to calculate the fee that need to be paid to the miner in order to execute operations. Of course, the cost of transactions can vary from one to the other depending on the details of the transaction and the structure and complexity of the smart contract. However, for a given smart contract C and a specific transaction t, one can perform certain calculations to compute the average cost or the maximum expected cost of the transaction in gas units, provided that the cost of each operation of the contract on the running blockchain system is known in advance. We defer the discussion of the computational details of gas usage of a given smart contract to Section 5.

On Plausible and Correct Patches. In this paper, we use the terminology of *plausible patch* and *correct patch*. Here we rely on the terminology in program repair literature (e.g. see [11]), where a correct patch is deemed to be correct via manual analysis, but a plausible patch is one produced by a repair technique since it passes all given tests. Since a formal complete specification of the intended program behavior is not available, the description of intended behavior given to a program repair technique is incomplete: it is given in the form of tests, assertions or vulnerabilities found. A plausible patch generated by the repair algorithm thus meets the intended behavior as per this incomplete description that was provided to the repair method. Thus, if the repair method was given a test-suite T, a plausible patch can still potentially fail a test t outside T. For this reason, a plausible patch cannot be guaranteed to be correct, and we need a manual validation step to ascertain how many of the plausible patches generated are correct. We have conducted such an evaluation in our work, in Section 7.

4 THE SMART CONTRACT REPAIR FRAMEWORK

In this section, we present a multi-objective genetic repair algorithm with mainly four objectives: two objectives related to the correctness of the smart contract and two related to the quality of the generated patch. We develop an efficient genetic search approach to generate a patch for a vulnerable smart contract. Our proposed genetic search technique employs mutation operators to generate fix candidates for the vulnerable contract and then uses fitness functions to evaluate the suitability of the candidate patch. The overall goal of our approach is to generate correct, high-quality, and gas-optimized fixes for the vulnerable smart contract.

Advantages of our search-based approach. The main motivation behind developing a genetic repair approach relies on the hypothesis that most software bugs introduced by programmers are due to small syntactic errors. Furthermore, the genetic search technique also has the following advantages with respect other common repair techniques.

- Semantic repair techniques employ symbolic execution and program synthesis for repairing programs. Employing such techniques for smart contract repair will deprive our approach of the natural ability to insert/delete statements which seems to be important for repairing common smart contract vulnerabilities like the reentrancy vulnerability.
- Template based repair techniques can be used as a purely static approach to smart contract repair. In this approach for every detected vulnerability type, a specific program transformation template can be employed for repair. Such an approach deprives us the possibility of

exploring a variety of patch candidates and enforce patch quality indicators in terms of gas consumption and patch simplicity.

4.1 Mutation Analysis of Smart Contracts

The mutation analysis of a vulnerable smart contract is the process through which a set of contract variants, called mutants, are generated by seeding a large number of small syntactic changes into the vulnerable contract using some mutation operators. The mutants are considered as patch candidates and we use the various correctness criteria and patch quality criteria, to choose and prioritize from the pool of patch candidates.

4.2 Mutation Operators and Patch representation

We employ three mutation operators. The *move* operator moves a given statement in the analyzed smart contract to some other location in the contract. The *insert* operator inserts a randomly synthesized statement before or after a given buggy statement. The *replace* operator replaces a potentially-buggy expression with another randomly synthesized expression. Our set of mutation operators contains both statement-level and expression-level operators to allow efficient mutation conducted at different granularity.

Patch Representation. A patch candidate is represented in terms of the mutation operations that need to be performed on the abstract syntax tree of the original vulnerable contract C being repaired.

4.3 Generating Mutated Smart Contracts

A large number of mutants may be introduced when repairing a vulnerable smart contract depending on the size of the contract, leading to searching among an extremely large set of mutants. Note that the validation process of the generated mutants can be extremely costly and time-consuming as also shown by other works on automated program repair [21]. Each mutant may need to be detected using the vulnerability detectors and tested against the original test suite. It is therefore necessary to apply a parallelization methodology in order to speed up the validation process of candidate mutants for a given vulnerable contract.

All mutation operators used in our repair framework can affect the cost of the vulnerable smart contract C which is also confirmed by our experiments. Their effect on the cost of the contract can be considerable, especially when the vulnerable contract contains loops that can be repeated a large number of times. If a plausible patch of C is obtained by replacing or inserting a statement within the body of the loop then the cost of the contract may change dramatically. It is crucial to search for a gas-optimized patch when repairing smart contracts in order to minimize the possibility of introducing new out-of-gas exceptions to the smart contract being repaired.

In general, generating a gas-optimized repair for a given vulnerable smart contract can be a computationally complex task. Note that the repair should not only fix the vulnerability in the contract but also needs to not increase the gas usage significantly. To achieve such a goal, one might choose to mutate the vulnerable smart contract C by favoring the mutation operators *move* and *replace* over the mutation operator *insert* when searching for low-cost patches. Indeed, intuitively, when we add new instructions onto the program would likely to increase the computational demand. Unfortunately, such a simple prioritization strategy does not necessarily lead to the least costly plausible patch for the vulnerable contract. Subtle interactions between the operators can turn a low-cost contract into a high-cost contract and vice versa. For example, the *insert* mutation operator which supposes to increase the cost of the contract by adding a new statement, may sometimes lead to a mutant with lower gas usage than the original vulnerable contract. Similarly, the *move*

```
1 bool a = true;
2 while (a) {
3     // Some computation
4 }
```

Fig. 2. Inserting `a = false` after line 1 reduces gas

```
1 int x = 0;
2 while (x <= 100) {
3     x = x + 2;
4     // Some computation
5 }
```

Fig. 3. Moving line 3 outside loop increases gas

mutation operator which supposes not to increase the cost of the contract can also lead to a mutant whose gas usage is higher than that of the original contact. The cost of the generated mutant does not depend only on the cost of the applied mutation operations but also on the way the operators change the behavior of the contract. We therefore cannot favor one operator over another when searching for low cost repairs without performing some analysis on the overall structure of the vulnerable contract.

Let us consider some trivial examples to demonstrate how the *insert* operator can turn a high-cost contract into a low-cost contract while the *move* operator may turn a low-cost contract into a high-cost contract. The program in Fig. 2 represents a buggy program. Suppose that we generate a mutant for this program by inserting a new statement after the initialization statement (line 1) of the form: `a = false;` . In this case, the loop in the generated mutant will be skipped and the average gas usage of the new mutated version will be much smaller than that of the original version. The program in Fig. 3 represents another potentially buggy program. Let us generate a random mutant of the program by applying the *move* operator so that the statement at line 4 (the loop counter update statement) is moved outside the loop. Obviously, this will turn the loop into an infinite loop and hence the contract will run out of gas after certain number of iterations. Note that since mutation makes random changes to the buggy smart contract, it may impact the performance and cost of the contract in many different arbitrary ways. This is critical especially when the buggy smart contract contains loops.

Observation 1. *There is insufficient information to predict the gas of a mutated contract by inspecting the mutation operations applied. For example, the successive applications of the mutation operators not introducing new statements* (*move*, *replace*) *does not necessarily lead to a low-cost mutant w.r.t. the original smart contract. Similarly, the successive applications of the mutation operator inserting new statements* insert *does not necessarily lead to a high-cost mutant w.r.t. the original smart contract. The cost of the generated mutants depends mainly on how the applied mutation operators change the behavior of the smart contract.*

As mentioned earlier, one of the biggest challenges that need to be addressed when using a genetic search approach for repairing smart contracts is how to speed up the generation and validation processes of mutated versions. We describe here a parallel search-based algorithm for efficiently generating patches. We assume here we have three versions of the mutate function: $mutateM(C)$ which mutates the contract C using only the *move* operator, $mutateR(C)$ which mutates the contract C using only the *replace* operator, and $mutateI(C)$ which mutates the contract C using only the *insert* operator. Since genetic repair approaches use mainly an exhaustive search algorithm to generate a patch, it is highly desirable to split the search space into sub-spaces. To do so, we use the mutate functions described above to split the search space into 7 smaller spaces as described in the following.

- $[SpaceS_1]$: this search space consists of the set of candidate patches that result from mutating the contract C using only the function $mutateM(C)$.

- $[SpaceS_2]$: this search space consists of the set of candidate patches that result from mutating the contract C using only the function $mutateR(C)$.
- $[SpaceS_3]$: this search space consists of the set of candidate patches that result from mutating the contract C using only the function $mutateI(C)$.
- $[SpaceS_4]$: this search space consists of the set of candidate patches that result from mutating the contract C using the two functions $mutateM(C)$ and $mutateR(C)$.
- $[SpaceS_5]$: this search space consists of the set of candidate patches that result from mutating the contract C using the two functions $mutateM(C)$ and $mutateI(C)$.
- $[SpaceS_6]$: this search space consists of the set of candidate patches that result from mutating the contract C using the two functions $mutateR(C)$ and $mutateI(C)$.
- $[SpaceS_7]$: this search space consists of the set of candidate patches that result from mutating the contract C using the functions $mutateM(C)$, $mutateR(C)$, $mutateI(C)$.

Note that for the effectiveness of the parallel algorithm we need to ensure that the search spaces are mutually-exclusive spaces so that no redundant mutants are generated and validated across various spaces. Recall that each mutant will be checked using the vulnerability detectors and against a set of test cases in addition to the gas usage requirement. Such validation process can be computationally complex specially when the search space of candidate patches is extremely large.

Mutants in S_7 are generated using the nesting operation $mutateX(mutateY(mutateZ(C)))$, where X, Y, and Z are distinct operators taken from the mutation domain $\{Move, Replace, Insert\}$. Assume $C_1 = mutateZ(C), C_2 = mutateY(mutateZ(C))$, $C_3 = mutateX(mutateY(mutateZ(C)))$. Then the validity function $V_{S_7}(C_3)$ for this search space S_7 can be formalized as follows.

$$V_{S_7}(C_3) = \begin{cases} Accept & \text{iff } diff(C_1, C) > 0 \wedge diff(C_2, C_1) > 0 \\ & \quad \wedge\ diff(C_2, C) > 0 \wedge diff(C_3, C_2) > 0 \\ & \quad \wedge\ diff(C_3, C) > 0 \wedge diff(C_3, C_1) > 0 \\ Reject & otherwise \end{cases}$$

Note that for a mutant to be added to the space S_7 it has to satisfy a somewhat complex condition. This is necessary in order to avoid overlaps with the other search spaces. Similar validity functions are defined for the other sub-spaces to ensure the mutually-exclusive property of the sub-spaces (please see Theorem 1).

Definition 2. *(**Properties of splitting strategy**). Let S be the search space of possible mutants of a vulnerable smart contract C generated using the operators* move, replace, *and* insert. *The splitting strategy of S into spaces $S_1, ..., S_7$ satisfies the following properties*

- *disjointness: for any two distinct sets S_i and S_j such that $(i, j = 1, ..., 7 \wedge i \neq j)$ we have $S_i \cap S_j = \emptyset$.*
- *completeness: $(S_1 \cup S_2 \cup ... \cup S_7) = S$.*

Theorem 1. *Spaces $(S_1, ..., S_7)$ are mutually exclusive spaces.*

Proof. (sketched). To prove the theorem we need to consider many different cases as we have 7 spaces. However, since the proof argument of all cases will be very similar and for brevity reason, we consider here only space S_7. For this case, we need to show that $S_7 \cap S_j = \emptyset \mid j = 1...6$. Hence, there are six possible sub-cases to consider. Recall that the mutants in S_7 are generated using the nesting operation $mutateX(mutateY(mutateZ(C)))$, where X, Y, and Z are distinct operators taken from the mutation domain $\{Move, Replace, Insert\}$. The theorem can be proven by contradiction.

- Let $S_i \cap S_7 \neq \emptyset \mid i \in \{1, 2, 3\}$. This implies that there exists a mutant m that belongs to both S_i and S_7. Note that since m belongs to S_i then it is generated using a single mutate function of the form $mutateX$, where $X \in \{Move, Replace, Insert\}$. It is easy to see then that

the mutant m cannot exist in the space S_7 as the addition of such mutant to S_7 contradicts with the definition of the validity function of the space S_7.

- Let $S_j \cap S_7 \neq \emptyset \mid j \in \{4, 5, 6\}$. This implies that there exists a common mutant m that belongs to both S_j and S_7. Note that since m belongs to S_j then m is generated from the nesting operation $mutateX(mutateY(C))$, where X and Y are distinct operators taken from the domain $\{Move, Replace, Insert\}$. Hence, the mutant m is generated using only two operators while ignoring the effect of one of the three operators. Therefore, the mutant m cannot exist in the space S_7 as this contradicts with the definition of the validity function of the space S_7 and the fact that mutants in S_7 are generated using the nesting operation $mutateX(mutateY(mutateZ(C)))$.

□

4.4 Parallel Repair Algorithm

We now describe a parallel genetic repair framework for vulnerable smart contracts. The repair framework consists mainly of eight processes running in parallel ($p_1||p_2||...||p_8$): the first seven processes ($p_1 - p7$) are responsible for generating compilable candidate patches of the given vulnerable smart contract corresponding to the search spaces ($S_1 - S_7$) and the last process (process p_8) is responsible for creating concurrent validation processes and selecting the most preferable patches generated as the base version to be further mutated. Such parallel repair framework would help to generate plausible repairs for vulnerable smart contracts in a much faster way than the repair framework that generates and validates candidate patches in a traditional sequential order.

```
1:  while p8 is running and space is not exhausted do
2:      C_base := Receives Base Contract from p8
3:      while Space S1 is not exhausted do
4:          C_new := mutateM(C_base)
5:          if C_new is compilable then
6:              Sends C_new to p8
7:              Break
8:          end if
9:      end while
10: end while
11: Terminate
```

Algorithm 1. Repair process p_1 in our algorithm (process $p_1, \ldots, p_7$ explores part of search space, $S_1, \ldots, S_7$).

Each process $p_i \mid i \in \{1, \ldots, 7\}$ is a long-running process. In each iteration, it waits for p_8 to send patch generation request. Upon the request is received with a base version of the vulnerable smart contract, the processes will then search for a compilable patch mutated from the received base version. The processes use mainly the set of mutation functions: $mutateM(C)$, $mutateR(C)$ and $mutateI(C)$ which mutate the vulnerable contract using some mutation operators. Every mutant is then be checked for their syntactic correctness via the use of a compiler. This technique has been shown very effective in early rejecting invalid patches. After the first compilable patch is generated, the process will then send it back to p_8 and wait for the next request. However, since the implementations of processes $p_1, \ldots, p_7$ are very similar, we present here the pseudo-code of one of them for brevity (we choose process p_1 that corresponds to the search space S_1). For readability, let us assume that we can get a fresh (new) mutant every time the function $mutateM(C)$ is used. The pseudo-code of p_1 is given in Algorithm 1. As one can see, Algorithm 1 can consider

```
1: Inputs : Vulnerable Contract C, Vulnerabilities U, Tests T
2: Inputs : Initial Population size IP, Generation size GR, Maximum Population size P_size
3: Inputs : Maximum Gas Usage Bound L
4: Output : Set of Plausible Patches
5: Patches := ∅
6: for i := 1; i ≤ IP; i := i + 1 do                    ▷ Each iteration executes in parallel
7:     C_new := Requests new mutant of contract C from p_1, ..., p_7
8:     C_new.fitness := Eval(C_new, U, T)
9:     Patches := Patches ∪{C_new}
10: end for
11: while (at least one of p_1, ..., p_7 has not terminated ∧ timeout not reached) do
12:     plausible := Filter_Plausible_Patches(Patches, U, T, L)
13:     if plausible != ∅ then
14:         return plausible
15:     end if
16:     Patches := NSGA2Selection(Patches, P_size)
17:     for i := 1; i ≤ GR; i := i + 1 do               ▷ Each iteration executes in parallel
18:         C_current_best := highest fitness patch from Patches
19:         C_new := Requests new mutant of C_current_best from p_1, ..., p_7
20:         C_new.fitness := Eval(C_new, U, T)
21:         Patches := Patches ∪{C_new}
22:     end for
23: end while
24: return ∅
```

Algorithm 2. Main Repair Algorithm (process p_8 which combines results from processes $p_1, \ldots, p_7$)

all possible combinations of random mutations of the function $mutateM(C)$ on the contract C until the corresponding patch space S_1 is exhausted.

We now discuss the implementation of the main process p_8 (Algorithm 2). This process takes as inputs: the original vulnerable smart contract C, the set of targeted vulnerabilities U, and the set of test cases T, then returns the patches that meets the quality requirements (plausible patches that pass given tests T and do not exhibit given vulnerabilities U). At the beginning, we conduct a population bootstrapping that a set of mutants is generated to have the initial set of mutants. The size of the set is controlled by the parameter IP (Initial Population Size). At the time new mutants should be generated, p_8 sends requests to the processes $p_1, \ldots, p_7$ (the **Requests** operation in Algorithm 2). Whenever one of the processes has generated a new compilable mutant, all other mutant generation processes will stop attempting to generate new mutants and the request is fulfilled. The *Eval* is used to calculate the fitness value of the patches. The objective functions are defined in Table 2. Note that all the objective functions are independent from one to the other, the *Eval* function therefore also issues new concurrent processes to speed up the patch fitness evaluation process. The control flow then enters the main loop. In each iteration, the algorithm first checks if there is already plausible patch existing in the maintained set of patches; this is accomplished by invoking the function *Filter_Plausible_Patches*. If it exists, this algorithm returns immediately the plausible patch. Otherwise, the maintain set of patches will be trimmed to the size P_{size} by the NSGA2 population selection algorithm[5] and yet another set of patches will be generated in the similar fashion. The base version used to generate the new set of patches is chosen to be the best patch among all the patches in the maintained set Patches. The evaluation of relative

quality between patches is based on their fitness values. In each iteration of the main loop, the number of new patches will be generated is determined by the parameter GR (Generation Rate).

We employ a timer in p_8 (not shown in pseudo-code for simplicity) which will be used to enforce termination of the process in case the time spent in the search process exceeds the bound *MaxBound*. The bound *MaxBound* should be chosen while taking into consideration the number of test cases, the size of the buggy program, and the estimated number of mutants in the search space assigned to the process. Note that processes work independently and terminate whenever a plausible patch is found or that the timer is fired.

Objectives or Fitness Functions. As mentioned earlier, the size of the search space can be extremely large even for programs whose source code size is small. Recall that the search space grows exponentially with the considered lines of code and hence the efficiency and performance of the genetic repair algorithm needs to be improved when examining candidate patches in the generated search space. While the parallel repair algorithm splits the large search space into smaller sub-spaces which improves considerably the patch generation process, the search sub-spaces can be still huge to be exhaustively explored in a reasonable time budget. The goal of the employed fitness functions is to guide the search towards plausible repair. We therefore integrate four fitness functions (objectives) with the patch generation process. The objectives are classified into primary objectives and secondary objectives. Primary objectives are related to the functional or correctness properties of the patch, while secondary objectives are related to the non-functional properties of the patch. The two main functional correctness objectives are the number of targeted vulnerabilities and the number of failing test cases. The number of targeted vulnerabilities can be retrieved from any smart contract vulnerability detector (e.g. Oyente [24]) while test cases can be provided by the vulnerable contract developers. The secondary properties or non-functional properties include the number of mutation operators applied on the generated patch and the gas usage or the cost of the patch. The designated fitness functions measure how many of desired functional and non-functional requirements a generated mutant meets. The mutation distance of the generated mutant from the original vulnerable contract is measured by counting the number of times the mutation operators applied to the generated mutant. This can be used to measure the simplicity of the generated mutant. The average gas usage is compared by the methodology described in section 5. The two secondary objectives are considered only when the generated patch is valid (fixed all targeted vulnerabilities and passes all test cases). Note that we give higher preference to a patch that fixes all detected vulnerabilities and passes all test cases with lower average gas usage and smaller number of syntactical changes w.r.t. the original vulnerable contract. We summarize these objectives (fitness functions) in Table 2.

Table 2. Objectives (fitness functions) used when generating patches

Description of objective	**Objective Purpose**	**Objective Type**	**Importance**
Number of targeted vulnerabilities	Patch correctness	Functional	Primary
Number of failing test cases	Patch correctness	Functional	Primary
Gas consumption	Patch gas optimization	Non-functional	Secondary
Mutation operation distance	Patch simplicity	Non-functional	Secondary

5 CHOOSING PATCH WITH LOWER GAS CONSUMPTION

One of the key challenges we encounter in this work is how to compare efficiently the average gas usage between the original contract and the repaired contract and how to compare the average gas

usage of different generated patches of a given vulnerable contract. In general, the gas cost of a smart contract depends on a number of parameters including memory cost, stack cost, and storage cost in addition to the instructions' costs. Hence, the gas consumption of a given path π in a smart contract *SC* can be a non-constant. It should be therefore described as a parametric formula that takes into consideration the parameters that affect the gas consumption of the path. We call the described parametric formula as *gas formula*.

To compare the average gas usage of two smart contracts, we propose the notion of *gas dominance*. The goal of the introduced gas dominance notion is to rank edited contracts (generated repairs of vulnerable contracts) based on their corresponding gas formula as an estimation on the relative average gas usage. This estimation is required as we cannot predict in advance the true average gas usage over their lifespan. Such a ranking approach can be used to select a low-cost repair for a vulnerable smart contract from the set of proposed repairs generated by the parallel repair algorithm.

5.1 The Gas Dominance Relationship

When formalizing the gas usage of smart contracts, we choose the specification of the gas cost function in the current Ethereum virtual machine specification (version EIP-150) [38] at the time of writing of this paper. From a high-level perspective, the gas usage of a single invocation to the smart contract depends on the user input to the smart contract, the blockchain environment, and the code of the smart contract. The gas usage of an execution (a transaction) to a smart contract is the sum of the gas usage of each executed instruction along the execution path. Formally, the gas cost function C of an instruction *inst* can be defined as

$$C(\sigma_{inst}, \mu_{inst}, I) = GU_{\text{OPCODE}_{inst}}(\sigma_{inst}, \mu_{inst}, I) + GU_{\text{New Memory}}(\sigma_{inst}, \mu_{inst}, I) \tag{1}$$

where $\sigma_{\mathbf{inst}}$ is the blockchain world state before the instruction *inst* is executed and $\mu_{\mathbf{inst}}$ is the machine state before *inst* is executed, the operation code $\text{OPCODE}_{inst} = I.code[\mu_{pc}]$ is a property of the execution environment I indexed by a program counter μ_{pc}, and $GU_{\text{OPCODE}_{inst}}$ is the gas formula associated to the operation code of *inst* and $GU_{\text{New Memory}}$ is the gas usage formula associated to the expansion of machine memory when executing the instruction *inst*. For more technical details about the definition of the gas cost function, we refer the reader to [38].

The total gas usage of an invocation (in the form of a single transaction) with the execution information specified in I can be defined as a gas function corresponding to the visited contract path triggered by the inputs:

$$GU_{\text{path}}(\sigma_p, \mu_p, I) = \sum_{inst \in \mathbf{Insts}} C(\sigma_{inst}, \mu_{inst}, I) \tag{2}$$

where $\mathbf{Insts} = (inst_0, inst_1, inst_2, \ldots)$ the sequence of instructions in the execution path determined by σ_p, μ_p and I, and $\sigma_p = \sigma_{inst_0}$, and $\mu_p = \mu_{inst_0}$. For a smart contract with k execution paths, we construct k gas usage functions, e.g. $GU_{path_1}, \ldots, GU_{path_k}$. We can then express the total gas usage of a smart contract *SC* over its lifespan as follows:

$$GU_{\text{lifespan, SC}} = \sum_{t \in trans} GU_{trans}(t)(\sigma_t, \mu_t, I_t) \tag{3}$$

where *trans* is the set of transactions to smart contract (denoted by *SC*) over its lifespan (the history of transactions of *SC*), and σ_t, μ_t and I_t are the world state, machine state and execution environment respectively when the first instruction of the invocation corresponding to transaction t was executed. We introduce a new higher order function GU_{trans} here that maps a transaction to

its corresponding gas usage function. Suppose the execution path is π for the transaction t, then $GU_{trans}(t) = GU_{path_\pi}$.

Given two repaired versions SC_a and SC_b for a vulnerable smart contract SC addressing the same vulnerabilities, we then favor the version with lower lifespan gas usage. However, since the future blockchain world state and the user inputs to SC can be of any possible combination which are generally unknown in advance, concrete lifespan gas usage of patched versions cannot be used to compare effectively the average gas usage of patches. We therefore propose to use what we call *gas dominance* as a method to compare the relative gas-efficiency between two patches by comparing the expected gas usage functions of them. So that for a given a smart contract SC_a with k execution paths, we can express the expected gas usage of SC_a as follows:

$$GU_E(SC_a)(\sigma, \mu, I) = \sum_{i=0}^{k} \mathbf{P}_i * GU_{path_i}(\sigma, \mu, I) \tag{4}$$

where $\mathbf{P}_i$ is the probability of $path_i$ being visited by an arbitrary execution of SC_a, GU_{path_i} is the gas usage function corresponds to program path $path_i$. For the cases where the contract paths invoke external functions, we need to include the gas usage introduced by the external function invocations in the equation of $GU_E(SC_a)$ of the contract.

Definition 3. *(**Gas Dominance Relation**). Given two smart contracts SC_a and SC_b, we say SC_a gas dominates SC_b (denoted by $SC_a >_g SC_b$) if and only if $GU_E(SC_a) \leq GU_E(SC_b)$ for all inputs and $GU_E(SC_a) < GU_E(SC_b)$ for at least one input to the smart contracts.*

Formally,

$$\begin{aligned} SC_a >_g SC_b \iff & \forall \sigma, \mu, I \; (GU_{E_a}(\sigma, \mu, I) \leq GU_{E_b}(\sigma, \mu, I)) \wedge \\ & \exists \sigma, \mu, I \; (GU_{E_a}(\sigma, \mu, I) < GU_{E_b}(\sigma, \mu, I)) \end{aligned} \tag{5}$$

where $GU_{E_a} = GU_E(SC_a)$ and $GU_{E_b} = GU_E(SC_b)$

The gas dominance relation has the following properties:

Property 1 (Irreflexive). *For all smart contracts SC, they do not gas dominate themselves. That is, SC must not gas dominate SC.*

Property 2 (Asymmetric). *For two arbitrary smart contracts SC_a and SC_b, if SC_a gas dominates SC_b, then SC_b must not gas dominate SC_a.*

Property 3 (Transitive). *For three arbitrary smart contracts SC_a, SC_b and SC_c, SC_a gas dominates SC_b and SC_b gas dominates SC_c, then SC_a must gas dominate SC_c.*

5.2 Lightweight Approximation for Determining Gas Dominance Relationship

In general, determining the gas dominance relationship between two smart contracts can be a computationally complex task and practically infeasible because the possible input space is generally too enormous. We therefore develop a lightweight approximation approach based on the notion of function dominance. We say that one gas formula dominates another formula if the magnitude of the ratio of the first formula to the second increases without bound as the inputs increase without bound. There are different ways to compare the gas consumption of two smart contracts and we describe here two approaches.

Given two contracts SC_a and SC_b, we first construct the expected gas usage formulas for SC_a and SC_b, namely $GU_E(SC_a)$ and $GU_E(SC_b)$. We then transform the equations $GU_E(SC_a)$ and $GU_E(SC_b)$ into polynomial expressions. Due to the fact that there might be terms containing non-polynomial functions, we use a *substitution mapping* to transform the gas formula into a polynomial expression. The substitution mapping is constructed as follows.

(1) For all monomial terms, they are unchanged.
(2) For other terms, the coefficient remains unchanged while the other parts of the term is mapped to a unique fresh variable.

All common non-monomial terms in $GU_E(SC_a)$ and $GU_E(SC_b)$ are mapped to the same fresh variable, that is, variable binding of the fresh variables are maintained for the substitution mappings e.g. if the formula $x^2 + sin(x)$ is substitution mapped to $x^2 + y$, the formula $x^2 + cos(x) + 3sin(x)$ should be substitution mapped to $x^2 + z + 3y$. A polynomial can be expressed as a sum of monomials where each monomial is called a term. The degree of the polynomial is the greatest degree of its terms. We denote the resulting polynomial equation for $GU_E(SC_a)$ by $GU_E^{poly}(SC_a)$ and the resulting polynomial equation for $GU_E(SC_b)$ by $GU_E^{poly}(SC_b)$. We then rearrange and simplify the resulting polynomial equations $GU_E^{poly}(SC_a)$ and $GU_E^{poly}(SC_b)$ as a sum of monomials. Let M_{SC_a} and M_{SC_b} be the sets of monomials in $GU_E^{poly}(SC_a)$ and $GU_E^{poly}(SC_b)$. We can determine the gas dominance relationship between SC_a and SC_b as follows (apply in order).

(1) If $|M_{SC_a}| \neq |M_{SC_b}|$, then SC_a and SC_b are not gas dominating each other.
(2) Let V_{SC_a} and V_{SC_b} be the vectors of coefficients of $GU_E^{poly}(SC_a)$ and $GU_E^{poly}(SC_b)$ respectively so that the order of elements of V_{SC_a} and V_{SC_b} should be aligned according to the same corresponding monomials.
   (a) If $V_{SC_a} \leq V_{SC_b}$ (all elements in V_{SC_a} are less than or equal to the corresponding elements in V_{SC_b} and $V_{SC_a} \neq V_{SC_b}$), then $SC_a >_g SC_b$.
   (b) If $V_{SC_a} = V_{SC_b}$, then SC_a and SC_b are not gas dominating each other.
   (c) If $V_{SC_a} \geq V_{SC_b}$ (all elements in V_{SC_a} are greater than or equal to the corresponding elements in V_{SC_b} and $V_{SC_a} \neq V_{SC_b}$), then $SC_b >_g SC_a$.
   (d) if none of the above conditions hold, then SC_a and SC_b are not dominating each other.

5.3 Integrating Gas Dominance Relationship into Genetic Patch Search Process

The above defined gas dominance relationship is for comparing the relative average gas consumption between two versions of the vulnerable contract. To enable the comparison among multiple patched versions of the original vulnerable contract, we here define the notion of *gas dominance level*, as defined in the following.

Definition 4. ***(Gas Dominance Level)**. Given a set of smart contracts, non-dominated sorting [5] is performed based on the gas dominance relationship. The gas dominance level of an arbitrary smart contract in the set is defined as its ranking in the non-dominated sorting result.*

The multi-objective genetic algorithm can now use the gas dominance level as one of the objectives, which serves to implicitly capture the effect of patches on the gas consumption (without having to compute the gas consumption directly).

5.4 Accelerating Gas Comparison by Generating Reduced Gas formulas

As described in the preceding, to compare the gas usage of two contracts we need first to synthesize gas formulas for the set of feasible paths in each contract. Note that the number of gas formulas generated for each patch can affect the computational complexity of the gas comparative approach dramatically. Suppose that the parallel genetic algorithm generates three plausible patches for a vulnerable contract C, namely C_1, C_2 and C_3. However, to compare efficiently the gas usage of the contracts C_1, C_2 and C_3 we only need to synthesize gas formulas for the set of different paths in the three contracts. It is sufficient to conduct a comparison between reduced versions of these contracts by skipping joint or common paths. This helps to reduce the computational complexity of the comparative approach.

Remark 1. *Syntactically identical paths among contracts share the same gas formula and therefore can be safely skipped during comparison.*

Definition 5. ***(Classifying paths in contracts)**. Let C be a vulnerable smart contract and $C^{'}$ be a repaired versions of C obtained by the parallel repair algorithm. A feasible path π in $C^{'}$ can be classified into one of the following categories*

- *π is a repaired path of some paths in C, or*
- *π is a new path w.r.t. the set of feasible paths in C, or*
- *π is a joint or common path between C and $C^{'}$.*

Note that a patch introduces to a given vulnerable smart contract may trigger a new set of paths that were infeasible in the original vulnerable smart contract. Thus, a repaired version of a contract may have new set of behaviors w.r.t. the original contract. This may happen for example when the patch updates an expression in a conditional statement in the original vulnerable contract. The advantages of distinction between the above three classes of paths are two-fold. First, it helps to reduce the number of paths that need to be considered when comparing the contracts and hence the number of gas formulas that need to be synthesized. Second, it helps to reduce the complexity of the final gas formulas of the contacts being compared. Note that since we use a genetic algorithm based on three mutation operators (*move*, *insert*, and *replace*), we can easily then classify paths in the contracts being compared into three categories: repaired paths, joint paths, or new paths. Typically, we can identify the locations of buggy statements in the contract and we can augment the repairing algorithm to label the locations of statements that have been influenced by the deployed patch. This facilitates the classification of paths in the generated repaired contract w.r.t. the original contract.

We now turn to describe an acceleration technique that can be applied before conducting the actual comparison between two similar contracts C and $C^{'}$. Let us denote the set of feasible paths in the two contracts by Π_C and $\Pi_{C'}$. The goal of the acceleration technique is to generate reduced versions of the contracts C and $C^{'}$ as follows:

(1) Compute the sets of paths that are unique in each contract as follows

$$Diff(C, C^{'}) = (\Pi_C \setminus \Pi_{C'})$$
$$Diff(C^{'}, C) = (\Pi_{C'} \setminus \Pi_C)$$

(2) Synthesize a gas formula for each path in the sets $Diff(C, C^{'})$ and $Diff(C^{'}, C)$ using Equation (2) and then compute the final gas formula by summing the resulting gas formula using Equation (4).
(3) Compare the resulting gas formulas using the comparative approach described at Section 5.

Comparing the gas usage of two contracts using their reduced versions (i.e., versions obtained by skipping joint paths or repaired paths whose gas formulas are equivalent) preserves soundness, as described in the following theorem.

Theorem 2. ***(Soundness of reduction)**. Let C be a vulnerable smart contract and $C^{'}$ be a repaired version of C. Let also $G(C)$ and $G(C^{'})$ be gas formulas for C and $C^{'}$ respectively and $G(C_R)$ and $G(C^{'}_R)$ be gas formulas for reduced versions of C and $C^{'}$ obtained as described at Section 5.4. $G(C_R)$ dominates $G(C^{'}_R)$ if and only if $G(C)$ dominates $G(C^{'})$.*

Remark 2. ***(Effectiveness of reduction)**. The accelerated comparative approach of smart contracts has lower computational complexity than the non-accelerated comparative approach. The amount of reduction on the computational complexity that can be obtained depends on the number of joint and*

repaired paths in the contracts being compared that can be skipped safely during the comparison (i.e., without adversely affecting the outcome of comparison).

The number of generated gas sub-formulas (for paths) and the complexity of the final gas formula (for the contract) can be significantly reduced if the acceleration approach is employed. This is crucial as synthesizing gas formulas for paths can be an expensive step specially for paths with cyclic behavior. Note that comparing reduced versions of contracts using simplified or reduced gas formulas that consider only different paths in the two contracts does not affect the soundness of the analysis. This is mainly due to the observation that only the set of different paths in the contracts can make the gas consumption of a contract dominates the other.

6 IMPLEMENTATION

In this section, we describe the implementation of the SCRepair tool, as well as the setup of the experimental evaluation (the results from the experiments will appear in the next section).

6.1 Prototype implementation

To evaluate our presented repair approach for vulnerable smart contracts, we have implemented a tool called SCRepair. The tool interacts and takes in inputs from the smart contract security analyzers Slither [8] and Oyente [24] in order to analyze and detect security vulnerabilities (if any) in the subject smart contracts. The tool Slither is a static analysis based detector which is able to reliably detect various vulnerabilities within a short time due to the lightweight nature of static program analysis. While being lightweight, it has been showing promising accuracy practically and used by the industry. On the other hand, the other supported vulnerability detector tool Oyente is a symbolic execution tool that works directly with Ethereum virtual machine code. It is able to detect many commonly occurring security flaws of Ethereum smart contracts. Notably, it can detect integer overflow and transaction order dependence vulnerabilities for which are difficult to be detected with pure static program analysis due to the need to reason about dynamic program behaviors. The fault localization information provided by the both vulnerability detection tools is used for fix localization in our search-based repair engine.

Since our repairing approach aims not only to fix the vulnerability but also to optimize the gas usage of the patched vulnerable smart contract, we have built a gas analyzer based on Oyente in a way such that it can generate the information for determining the approximate gas dominance relationship. For determining gas dominance, it has a gas usage model extended from the Oyente implementation which is closer to the actual Ethereum virtual machine's gas model.

In Fig. 4 we give the schematic diagram of our smart contract repair tool in which we describe the main components of the tool. The tool consists of five units: the vulnerability detector, the test case executor, the gas ranker, the patch generator, and the main process controller. Units except the main process controller are executed in worker processes/threads. The main process controller unit manages all the worker processes and threads to perform the repair task in a concurrent manner.

We have also implemented a variant of SCRepair with an unguided random search repair algorithm called SCRepair-URS. This implementation mostly reuses the SCRepair implementation except that the genetic search mechanism is removed. The patch evaluation can now be terminated early as long as it has sufficient information to assert that the patch under validation is not plausible (*e.g.*, as soon as a vulnerability is detected or a test case failed under the patched version). This acts as an optimization which is not possible to apply to genetic repair algorithm since early termination does not give the algorithm an evaluation of the fitness functions (*e.g.*, does not generate total number of test cases failed).

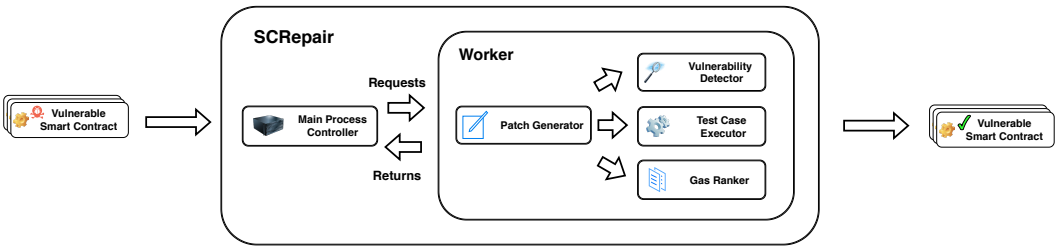

Fig. 4. The schematic diagram of the SCRepair tool

6.2 Etherscan Vulnerable Dataset (EV-DS)

To evaluate our repair approach, we have constructed a dataset of vulnerable smart contracts taken mainly from Etherscan as a proxy to real-world deployed smart contract source code. Etherscan is a well-known block explorer, search, API and analytic platform for Ethereum mainnet, which is the main network wherein actual transactions of smart contracts take place on a distributed ledger. A large amount of information related to the smart contracts can be extracted from Etherscan, e.g., deployment address, verified source code, byte-code and application binary interface (ABI) of deployed contracts. From Etherscan we first collected all 38,225 available smart contract source code files which correspond to smart contracts deployed before 1st August 2018. After de-duplication, 34,400 files remained. These source files are then analysed using the tool Oyente. We obtained 2,752 vulnerable smart contracts with different types of vulnerabilities. Four types of vulnerabilities have been detected on this dataset: transaction order dependence (TOD), reentrancy (RE), exception disorder (ED), and integer overflow (IO).

The TOD happens when the user of a smart contract assumes a particular state of a contract, which may not exist when his transaction is processed potentially leading to malicious behavior. Reentrancy vulnerability is probably the most widely known vulnerability as it led to the infamous DAO attack (causing a loss of 60 million US dollars in June 2016). RE happens when a contract is called by another contract so that the original contract has to wait for the call to finish. This intermediate state can be exploited. The contracts may suffer also from the so-called exception disorder (ED) vulnerability where the contract does not check explicitly whether the send operations have been completed successfully. Integer overflow (IO) is a common problem across all systems which could be used to modify the program state in an unwanted manner by deliberately providing large numbers as inputs leading to incorrect results being calculated in mathematical operations.

While the selection of smart contracts shown in Tables 3 and 4 has been made randomly from the dataset EV-DS, we have considered some key criteria when selecting these smart contracts. The two main criteria we considered are: (1) the size and complexity of the vulnerable smart contract measured mainly in terms of the number of lines in the contract, and (2) the popularity and the number of available transactions of each vulnerable smart contract. Therefore, we filter the dataset according to the following rules before random sampling:

(1) Number of lines of code (exclude comments) > 30
(2) Number of transactions (at the time data collection was performed) > 30

6.3 Test Case Generation for EV-DS Dataset

Since test cases for smart contracts are in general not available on the blockchain and the authors of the deployed smart contracts are also not contactable [24], we therefore use a novel method to generate regression test cases from the available transactions to the subjects smart contracts on

Table 3. EV-DS dataset subject smart contract information

Name of Contract	# Lines	# Transactions	# Regression Tests	Supported by SCRepair
Autonio ICO	330	34	31	Yes
Airdrop	62	147	7	Yes
Banana Coin	117	360	24	Yes
XGold Coin	272	308	304	Yes
Flight Delay Issuance	429	80	1	No
Hodbo Crowdsale	268	36	18	Yes
Lescoin Presale	351	115	107	Yes
Classy Coin	217	574	495	Yes
Yobcoin Crowdsale	481	515	435	Yes
Classy Coin Airdrop	49	137	4	Yes
OKO Token ICO	232	179	173	Yes
ApplauseCash Crowdsale	407	43	42	Yes
HDL Presale	239	94	93	Yes
Privatix Presale	179	78	11	Yes
MXToken Crowdsale	186	56	37	Yes
EthereumFox	77	493	491	No
dgame	42	302	108	Yes
Easy Mine ICO	351	1339	491	Yes
Siring Clock Auction	978	1641	2	Yes
Government	83	502	366	No

the blockchain. For every transaction (denoted by t) to the subject smart contract, we capture the inputs and the changes to the blockchain state during the execution of t which then considered as the inputs and expected behaviors of the generated regression test case. A generated regression test case for a transaction t contains the following elements:

(1) Blockchain state before executing the transaction t.
(2) The function being invoked and the corresponding argument values.
(3) Blockchain state after executing the transaction t.
(4) The return values of invoked functions.

However, as the whole blockchain state can be very huge (in the magnitude of terabytes), it is impractical to simply store relevant versions of the blockchain state. To address this issue, we only capture relevant states of the Ethereum accounts in the blockchain before and after the execution of the transaction t. The generation of each regression test case is then run against the original vulnerable smart contract to check the validity of the newly generated test case. During the test case generation process, we have set a timeout bound of 5 minutes for the execution time of each regression test case. Regression tests requiring longer time are terminated and discarded. Table 3 shows the number of regression test cases generated for each subject contract. The generated regression test cases are then used in the automated repair experiments.

6.4 Factors Affecting our Repair Algorithm

Before discussing the research questions that we developed to evaluate the presented genetic repair algorithm, we first summarize the key factors that affect the correctness and efficiency of our genetic repair algorithm.

(1) **Quality of test suite and vulnerabilities**. The quality of provided test suite for a given vulnerable smart contract has a major impact on the genetic repair algorithm. Recall that a mutant is considered as a repair when all available test cases pass and the vulnerability detector does not report any vulnerability found. In our experiments, we constructed the test

suite with a script to convert past block-chain transactions as positive test-cases as described in subsection 6.2. The vulnerabilities detected by a smart contract checker like Oyente and Slither constitute the negative behavior that the generated patches should avoid.

(2) **Timeout allocated to the algorithm**. A feasible exploration of the search space (candidate patches) depends heavily on the amount of resources allocated to the genetic algorithm. In general, the size of the generated search space of a given vulnerable contract depends on multiple factors including: (i) the size and complexity of the contract being repaired, (ii) the number of buggy statements in the contract, and (iii) the mutation operators used by the algorithm. However, the number of mutants that can be examined during the search is limited to the time budget allocated to the algorithm. The bigger the time budget, the higher the probability to produce a plausible patch.

(3) **The consideration of gas consumption of patches**. Considering the gas when searching for plausible patches of a vulnerable smart contract can be of great benefit. First, it can help to generate a low-cost repair for a given vulnerable smart contract by comparing the gas consumption of generated patches and selecting the one with low average cost. Second, it can be used to optimize the efficiency of the genetic search algorithm in various ways. For example, it can be used to detect and discard infeasible patches early. Note that a patch can be a plausible patch (passes the test-cases) but infeasible to be deployed on a real blockchain. This happens when the generated patch consumes a significantly large amount of gas and thus leads to expensive transactions. To reduce the computational complexity of the algorithm, one might need to maintain during the genetic search the best known low-cost average gas usage (let us call it g_{max}) of a plausible patch. Then when a new plausible patch is found that has lower average cost, the bound g_{max} will be updated accordingly. The bound g_{max} can be updated on-the-fly during the search and used to discard infeasible patches early without necessarily examining the entire test suite.

(4) **The number of genetic mutation operators used by the algorithm**. Note that the size of the search space that needs to be examined when searching for a plausible patch for a vulnerable contract can be extremely large. Recall that the search space of a given vulnerable smart contract is generated by mutating (buggy) statements in the contract. Hence, the size of the generated search space grows exponentially w.r.t. the number of considered lines in the contract and the number of mutation operators. The smaller the number of the mutation operators, the smaller the size of the search space and the faster the algorithm. However, reducing the number of the mutation operators may reduce significantly the capability of the algorithm to produce plausible patches.

(5) **The state space search order**. As the search space grows, the organization of mutants or candidate patches into sub-spaces becomes more critical to the efficiency of the algorithm. In general, there is no specific search strategy that one can follow when examining the candidate patches of a given vulnerable smart contract. The search can be purely sequential and random or it can be parallelized based on the semantics of the mutation operators. However, as expected, the search can be optimized by taking into consideration some interesting factors including the semantics of the bug, the semantics of the mutation operators, and the gas consumption of generated patches.

As one can see from the aforementioned factors, the correctness and efficiency of the genetic algorithm can be evaluated under many different settings. For example, one might wonder how does the algorithm perform when enabling/disabling the gas calculation of generated patches or when increasing/decreasing the size of test suite or the amount of time budget allocated to the genetic algorithm. In this work, we choose to evaluate the correctness and efficiency of the

Table 4. Results for RQ1, showing efficacy of patching.
(Timeout is 1 hour, OOM denotes out of memory)

Name of Contract	Vulnerabilities Discovered	SCRepair Vulnerabilities Repaired (Correct/Plausible)	SCRepair-URS Vulnerabilities Repaired (Plausible)	Avg Run Time (mins) SCRepair/ SCRepair-URS
Autonio ICO	ED(1)	ED(0/1)	ED(1)	3/0.4
Airdrop	ED(4)	ED(3/4)	None	8/OOM
Banana Coin	ED(1), RE(1)	ED(1/1), RE(1/1)	ED(1), RE(1)	16/15.8
XGold Coin	ED(2)	ED(2/2)	None	12/60
Hodbo Crowdsale	ED(2)	ED(2/2)	ED(2)	22/19.8
Lescoin Presale	ED(2)	ED(1/1)	ED(2)	2/14.8
Classy Coin	ED(1), RE(1)	None	None	29/60
Yobcoin Crowdsale	ED(2), RE(1)	ED(1/1), RE(1/1)	None	60/OOM
Classy Coin Airdrop	ED(2)	ED(1/2)	ED(2)	1/3
OKO Token ICO	ED(4), RE(2)	ED(1/1), RE(1/2)	None	60/OOM
ApplauseCash Crowdsale	ED(2), RE(1)	ED(1/1)	None	60/OOM
HDL Presale	ED(3)	ED(3/3)	None	55/51.8
Privatix Presale	ED(1)	ED(1/1)	ED(1)	2/2.8
MXToken Crowdsale	ED(1)	ED(1/1)	ED(1)	30/5.2
dgame	IO(3), TOD(1)	None	None	2/OOM
Easy Mine ICO	IO(6), TOD(1)	None	None	60/50.4
Siring Clock Auction	IO(3)	IO(0/1)	None	4/60
Total	ED(28),IO(12),RE(6) TOD(2), sum: 48	ED(18/21),RE(3/4) IO(0/1), sum: 21/26	ED(10),RE(1) sum: 11	

genetic algorithm by considering five key *research questions*. The goal of the research questions is to evaluate the presented parallel genetic repair algorithm and to understand and draw conclusions about the factors affecting the correctness and quality of generated patches.

7 RESEARCH QUESTIONS AND EXPERIMENTAL RESULTS

Responsible Disclosure. We decide to publish the dataset for open science. The blockchain system is decentralized so even if we want to contact the owner we cannot find them.

Experimental Setup. We run our tool on a single Amazon AWS EC2 instance c5.24xlarge which has 192GB of RAM and AWS-customized 2nd generation of Intel Xeon Scalable processor with 96 CPU execution threads allocated. Our repair algorithm and its implementation can be run on a compute cluster of multiple computing nodes. However, we run our experiments on a single node in this work for the simplicity and sake of financial budget. Among the 20 vulnerable smart contract subjects, our implementation prototype was able to handle 17 of them. The remaining 3 have syntax constructs that are currently unsupported or the version of Solidity used in the implementation of these contracts is too old to be supported. Therefore, we carried out our experiments on the 17 supported subjects. For increasing the variety of vulnerabilities being considered while avoiding the expensive cost of symbolic execution, we have employed Slither as the vulnerability detector for the first fourteen subjects and Oyente for the remaining subjects. We limit the scope of targeted vulnerabilities in our experiments to have more focused study to the following vulnerabilities: ED, RE, IO, TOD. Our implemented gas analyzer is employed for determining the gas dominance relation between patches. In our experiments, the maximum gas usage bound L is not specified since a reasonable value is subject to the concrete usage of the subject smart contracts from the viewpoint of the original developers.

RQ1: How effective is the repair algorithm at fixing detected bugs?

Setup. To demonstrate the effectiveness of the presented genetic repair algorithm in fixing vulnerable smart contracts, we run the genetic algorithm on the selected set of smart contracts. We

evaluate the effectiveness of the algorithm by measuring the number of vulnerabilities that can be detected and repaired correctly and the time it takes to generate correct patches of these vulnerable contracts. Recall that a repair is generated by the algorithm when all test cases pass and no targeted vulnerability is found. We call such as a fix as a plausible fix. Hence, the generated patch might still not be a correct patch. We then check the correctness of the generated patches by inspecting the semantics of the patches manually. We assert a plausible fix for one vulnerability as correct if it repaired the vulnerability being detected while the original business logic is not modified and the fix does not introduce new features or vulnerabilities to the code. We use the unguided random search implementation as the baseline to evaluate the effectiveness of the designated guidance in the search process. This is essential since the expensive complete patch quality assessment by the objective functions could result in lowering the efficiency and effectiveness[2, 33].

Results. For each of the considered vulnerable contracts, we have run our algorithm five times, each time with a timeout of one hour. We report the average value of the run time and the sum of plausibly successfully repaired vulnerabilities among five runs as the final results. Table 4 shows the summary of the results and the average run time of the algorithm. The algorithm was able to plausibly repair 26 occurrences of vulnerabilities among the 48 detected vulnerabilities. The average run time of the algorithm over the considered 17 subjects was 25 minutes. We noticed that the main bottleneck of the implementation is due to the test case execution time which often consumes the most computational resources and blocks the synchronization barrier of each iteration of the main loop of the algorithm. When inspecting the generated patches, we found that our algorithm was able to fix correctly 21 vulnerabilities out of the detected 48 vulnerabilities. With the same timeout, our genetic algorithm was able to plausibly fix 15 more vulnerability than the unguided random search version yielding a 136% improvement. This clearly shows the guidance in the search process from the genetic algorithm has increased the repair efficiency significantly. Moreover, a careful inspection of the results reported in Table 4 leads to the following interesting observations.

Observation RQ1.1. *As shown in Table 4 there are four different classes of vulnerabilities that have been considered when evaluating the algorithm, namely, ED, RE, IO, and TOD. We observed that most of the vulnerabilities of the classes ED and RE have been fixed correctly by the algorithm, where 21 out of the 28 detected EDs have been plausibly repaired and 4 out of the 6 detected REs have been plausibly repaired. On the other hand, the algorithm was unable to generate correct patches for any of the vulnerabilities of the classes IO and TOD; one plausible patch for IO was generated.*

Observation RQ1.2. *The occurrence rates of the vulnerabilities ED, RE, IO, and TOD in the considered vulnerable contracts are as follows: ED occurs* 58%*, RE occurs* 13%*, TOD occurs* 4%*, and IO occurs* 25%*. We observed that the ED vulnerability is the most frequently occurring class of bugs in the selected vulnerable contracts, where 28 out of the 48 detected bugs are ED bugs.*

Observation RQ1.3. *We observed that 7 out of the considered 17 vulnerable contracts have been repaired in less than 10 minutes, where most of these contracts contain multiple bugs. This demonstrates clearly the efficiency of the presented parallel genetic repair algorithm in fixing vulnerabilities in a considerably short amount of time.*

Answer to *RQ1*: Among the 48 detected vulnerabilities in the 20 vulnerable smart contracts, the algorithm was able to fix plausibly 26 vulnerabilities, where 21 of these plausible fixes have been verified to be correctly fixing the vulnerabilities. Notably, our implementation fully repaired 10 of the 20 contracts.

Table 5. Results for RQ2, showing gas variation between buggy contract and patched versions.

Name of Contract	# plausible patches	# patches with diff. gas formula from original	# patches with diff. gas dominance level from original	Ratio of plausible patches yielding diff. average gas
Autonio ICO	7	7	6	85.7%
Airdrop	5	0	0	0%
Banana Coin	4	4	0	0%
XGold Coin	7	7	0	0%
Hodbo Crowdsale	3	3	3	100%
Classy Coin Airdrop	5	5	5	100%
HDL Presale	1	0	0	0%
Privatix Presale	9	8	8	88.89%

RQ2: Does fixing the vulnerability affect the gas usage?

Setup. When fixing the detected vulnerabilities, expressions in the vulnerable smart contracts will be modified. However, it is unclear whether plausibly fixing the vulnerabilities would change the average gas consumption of the smart contract. We therefore perform a comparison on the average gas consumption between the original vulnerable smart contract and the plausibly patched versions generated from five repeated runs conducted in RQ1. Gas dominance levels between the original contract and the patched versions are computed. We assert the patched version has different average gas consumption from the original version when they are of different gas dominance levels. To calculate the gas dominance level, the gas formula of the original version and the patched versions will be generated, as described in earlier sections.

Results. Table 5 shows the difference in average gas consumption between the plausible patches and the original version. Subjects for which plausible patches could not be generated within time limit (1 hour) are omitted for consideration of this RQ. To sum up, 6 out of 8 (75%) of our set of selected subjects have plausible patches with gas formula that are different from the original vulnerable version while half (50%) of our set of selected subjects have plausible patches having gas dominance levels different from that of the original vulnerable version. This suggests the possibility that fixing vulnerabilities in smart contracts can change the average gas consumption of the original contract. For the subjects with plausible patches amending the average gas consumption, each independent patch generation process has high probability (93.65% in our experiments) of generating plausible patches of gas dominance levels different from the original version.

> Answer to *RQ2*: In general, when fixing vulnerabilities in a vulnerable smart contract, the gas should be one of the factors considered in the repair process.

RQ3: Can plausible patches vary significantly in average gas consumption?

Setup. Further, we would like to investigate whether there is a possibility to plausibly fix the vulnerabilities with more than one patch yielding to different average gas consumption across patches. In other words, we intend to understand whether the same bugs can be fixed with patches of different average gas consumption. If the answer is positive, we then justify the need to attempt pursuing a more gas-efficient plausible patch during the search process. We conduct our analysis on the patches generated in RQ1 across five repeated runs. We leverage gas dominance levels of patches as a proxy to compare the difference in average gas consumption between patches. We assert a patched version has different average gas consumption from the other when they are of different gas dominance levels. Note that two patched versions have different gas dominance levels

Table 6. Results for RQ3, gas variation among patch candidates is shown.

Name of Contract	# plausible patches	# unique gas formula among patches	# gas dominance levels among patches
Autonio ICO	7	7	7
Airdrop	5	1	1
Banana Coin	4	2	1
XGold Coin	7	5	1
Hodbo Crowdsale	3	2	2
Classy Coin Airdrop	5	1	1
HDL Presale	1	1	1
Privatix Presale	9	3	3

when the gas formula of one of the two versions dominates the other. However, to calculate the gas dominance level of generated patches, the gas formulas of the original version and the patched versions need to be generated first.

Results. Table 6 shows the difference in average gas consumption between the generated plausible patches of selected vulnerable contracts. Subjects for which plausible patches could not be generated within time limit (1 hour) are omitted for consideration of this RQ. For 5 out of 8 subjects (62.5%), we were able to get a set of plausible patches with more than one corresponding unique gas formulas, indicating the diversity of gas consumption between plausible patches addressing the same set of vulnerabilities. We noticed that plausible patches have around two gas dominance levels among them, on average, for a given contract.

Observation RQ3.1. *For 62.5% of the considered subjects, there exist plausible patches having different average gas consumption.*

> Answer to *RQ3*: Different plausible patches can yield various average gas consumption for fixing the same vulnerabilities. We should therefore attempt to guide the search towards more gas-efficient plausible patches besides considering their correctness.

RQ4: How effective is the gas ranking approach at producing low-cost patches?

Setup. During the patch generation process, we have integrated our proposed gas comparison approach to compare the relative gas usage of generated patches. The relative gas dominance relationship is then used in the genetic patch generation process as a guidance to generate a potentially gas optimized patch. To evaluate systematically the effectiveness of the gas usage objective in producing low-cost patches, we run our repair algorithm on the selected vulnerable smart contracts under two different settings: the first setting is when the the gas ranking objective is active (done in RQ1) and the second setting is when the gas ranking objective is deactivated. The first setting is a reuse of patches generated in RQ1 while the second setting is additional runs with repeating factor of five and timeout of one hour. Later, we run all patches generated in both settings on our generated test cases and collect the average runtime gas usage of each setting. For consistent and fair comparison, we only consider patches fixing all vulnerabilities. Different from RQ2 and RQ3, this RQ attempts to expose the change in average gas consumption for the previous usages of the contracts to infer practical gas cost changes.

Results. Table 7 shows the summary of average gas usage of patches generated with and without the gas objective being activated. Subjects that plausible patches could not be generated within

Table 7. Results for RQ4, showing the average gas usage of the patched versions (on the given tests) in two settings. The percentage shows the improvement on average gas usage when the gas objective is enabled.

Name of Contract	Average gas usage (Gas objective is enabled)	Average gas usage (Gas objective is disabled)
Autonio ICO	87092.2 (0%)	87092.2
Airdrop	73633.4 (0.92%)	74316.1
Banana Coin	72535.1 (0.01%)	72542.3
XGold Coin	**46154.6 (6.37%)**	**49296.3**
Hodbo Crowdsale	38848.3 (0%)	38848.6
Classy Coin Airdrop	72810.5 (0%)	72810.5
HDL Presale	48536.6 (0%)	48536.525
Privatix Presale	**40323.7 (9.31%)**	**44464.46**
MXToken Crowdsale	43247.4	No patch generated

time limit (1 hour) are omitted for consideration of this RQ. Overall, 6 out of 8 subjects among subjects for which both settings can generate plausible patches (75%), the gas objective is effective to reduce the average cost of the patches by up to 9.31% for our subjects. Two subjects (Autonio ICO and Classy Coin Airdrop) do not have varied average gas usage between patches generated in two settings. One subject (*MXToken Crowdsale*) does not have plausible patch generated where the gas objective is deactivated in the five repeated runs. In addition, we have also done careful profiling of the algorithm exposing the fact that gas ranking has frequently been the determining factor of patch rankings during the repair process of the selected subjects even though the gas objective is employed as a secondary objective.

Answer to *RQ4*: When enabling the gas objective during repair, we observed that the average gas consumption of generated patches of four vulnerable contracts has been reduced comparing to the setting in which the gas objective was disabled. We observed also that the average gas of two subjects has been considerably reduced when enabling the gas objective, where the average gas of the patched version of XGold Coin contract is reduced by 6.37% and the average gas of the patched version of Privatix Presale contract has been reduced by 9.31%. This is a considerable reduction as gas costs real money.

RQ5: How does the time budget impact our effectiveness at fixing bugs?

Setup. Allocating or estimating a feasible time budget to a genetic repair algorithm is an interesting open problem. It is crucial as it affects the capability of the algorithm in generating plausible patches for a given vulnerable contract. There are some key factors that should be taken into consideration in order to allocate a feasible time budget to our repair algorithm including: (i) the size of the test suite, (ii) the complexity of the contact (i.e., larger contracts may take longer time to be analyzed than smaller contracts), and (iii) the estimated size of the search space which in turn depends on the number of the mutation operators used by the algorithm and size of the original vulnerable contract. To address this research question, we choose to evaluate the algorithm under two different time budgets: the first is when we set the timeout to 30 minutes and the second is when we set the timeout to one hour. The goal is then to measure the number of vulnerable contracts that have been repaired under the two settings.

Table 8. Results for RQ5, obtained by varying the timeout from 30 minutes to 1 hour

Name of Contract	Vulnerabilities Discovered	Vulnerabilities Plausibly Fixed (30mins/1hr timeout)
Autonio ICO	ED(1)	Same
Airdrop	ED(4)	Same
Banana Coin	ED(1), RE(1)	Same
XGold Coin	ED(2)	Same
Hodbo Crowdsale	ED(2)	Same
Lescoin Presale	ED(2)	ED(0/1)
Classy Coin	ED(1), RE(1)	Same
Yobcoin Crowdsale	ED(2), RE(1)	ED(0/1), RE(0/1)
Classy Coin Airdrop	ED(2)	Same
OKO Token ICO	ED(4), RE(2)	ED(0/1), RE(0/1)
ApplauseCash Crowdsale	ED(2), RE(1)	ED(0/1), RE(0/0)
HDL Presale	ED(3)	ED(1/3)
Privatix Presale	ED(1)	Same
MXToken Crowdsale	ED(1)	ED(0/1)
dgame	IO(3), TOD(1)	Same
Easy Mine ICO	IO(6), TOD(1)	Same
Siring Clock Auction	IO(3)	Same

Results. Table 8 shows the results of running the algorithm over the selected vulnerable smart contracts using two different values of the timeout parameter (30 minutes and 1 hour). As shown in the table, when setting the timeout parameter to 30 minutes the algorithm was able to generate plausible patches for 17 vulnerabilities out of the 48 detected ones, achieving a success rate of 35.4%. On the other hand, when setting the timeout parameter to 1 hour the algorithm was able to generate plausible patches for 26 vulnerabilities, achieving a success rate of 54.2%. While the amount of improvement on the repair rate looks somewhat small, it is very crucial as it shows that some vulnerabilities can be only repaired when increasing the timeout to 1 hour. This clearly demonstrates the impact of the timeout parameter on the effectiveness of the algorithm. However, since every detected vulnerability in a given vulnerable smart contract needs to be repaired and the fact that the size of the search space can be extremely large, the time budget allocated to the algorithm can play a key role in the successful termination of the algorithm. When we increase the time budget of the algorithm, we increase the size of the explored search space which in turn increases the probability of generating plausible patches.

Answer to *RQ5*: When we increase the timeout parameter of the algorithm from 30 minutes to 1 hour we observe that the vulnerability repair rate of the algorithm has been increased from 35.4% to 54.2%, where the genetic algorithm was able to repair 9 extra vulnerabilities. This demonstrates clearly the importance of allocating a substantial time budget (at least one hour) to the algorithm when repairing vulnerable smart contracts.

7.1 Threats to Validity

Finally, we discuss the threats to validity of our experimental results.

Internal validity. Threats to internal validity are related to the representative nature of our conclusions and summaries made based on our experiment results. In our experimental study, we have conducted our experiment on a sample dataset to evaluate our approach. The size of

the dataset is however limited since this is the first automated smart contract repair work, and therefore, there is no consolidated dataset for use like Detect4J[17] for Java. We are aware that our approach employs biased random search techniques, and for this reason each experiment was repeated five times. We admit that the presented results are potentially skewed even though we have conducted our experiments with a replication factor of five times for each setup.

External validity. External validity treats are related to the ability to generalize our findings. We have only evaluated our work on four known vulnerability types. While our approach is vulnerability-agnostic, the efficacy in terms of fixing other vulnerabilities remains unknown. On the other hand, we have conducted our experiments on real-world subjects as an attempt to investigate the performance of approach. This does not guarantee that similar efficacy will be exhibited for arbitrary vulnerable smart contracts.

8 RELATED WORK

We discuss the related literature on automated program repair, smart contract analysis, and gas usage calculation of smart contracts.

8.1 Automated Program Repair

Automated program repair [11] has been the subject of considerable recent attention in the software engineering research community. Commonly, they attempt to automate the process of fixing the bugs exposed by failing test cases, and these techniques are collectively called *test-based repair techniques*. The patch that can fix all the given tests is called a *plausible patch*. Several test-based program repair approaches have been developed. These approaches can mainly be classified into search-based and semantics-based approaches.

Search-based approaches developed by [21], and [26] show promising results towards the automation of bug fixing. The key idea of their approaches is to use failing test cases to identify bugs and then apply mutations to the source code until the program passes all failing test cases, while continuing to pass previously passing tests. Genetic programming [21] as well as random search [33] have been used as search techniques for finding a plausible patch, a patch passing given test-cases. GenProg [21] is one of the early works among search-based repair techniques.

Semantic analysis techniques like SemFix [29], Nopol [40], DirectFix [27], SPR [23], Angelix [28] and JFIX [19] split patch generation into two steps. First, they infer a desired specification (or a repair constraint) for the buggy program statements, which is often accomplished via symbolic execution of the given tests. Second, they synthesize a patch for these statements based on the inferred specification, using program synthesis techniques. These works view program repair as a specification inference problem, as opposed to searching among candidate patches. These approaches can be combined with search: we explore patches by considering insert/delete/replace of statements, while the semantic analysis can help synthesize expressions to be inserted in the statement replacements [41].

Apart from automated program repair approaches driven by functionality (often exposed by a test-suite), some other studies e.g. Caramel [30] attempts to automatically fix non-functionality bugs (such as performance bugs). Such bugs can be fixed by inserting an early termination statement inside loops. The generated patch can potentially reduce the run time of the program.

Our smart contract repair problem (defined in Problem 1) is similar to the test-based program repair problem. We also leverage the test cases to examine functional correctness of patches. However, since vulnerabilities in smart contracts have been raising serious financial losses, our patches need to not only be test-adequate but also secure and gas-aware.

8.2 Testing and Analysis of Smart Contracts

Analysis of smart contracts is a popular topic that has received a lot of attention recently, with numerous tools being developed based on fuzz testing, symbolic execution and constraint solving. [1, 14, 16, 18, 24, 37]. Oyente [24] is one of the earlier works on symbolic analysis of smart contracts. The work in [1] translates smart contract source code to Isabelle/HOL in order to validate smart contracts. The authors use the symbolic security analyzer Oyente [24] to detect vulnerabilities in smart contracts. The tool ContractFuzzer [16] uses fuzz testing to detect security vulnerabilities in smart contracts. Recently, van der Meyden [37] conducted a formal analysis of an abstract model of smart contract code (atomic swap smart contracts) using the epistemic MCK model checking tool [10]. He showed how to automatically verify that a concrete implementation of atomic swap satisfies its specification using epistemic-temporal logic model checking.

There is also a considerable amount of work on the mutation testing of smart contracts [9, 15, 39]. Mutation testing [31, 32] is a technique for evaluating the quality of a set of test cases (i.e., a test suite). It works by introducing faults into a system via source code mutation and then analyzing the ability of some developed test suite to detect these faults. The work in [39] has implemented certain mutation operators and tested them on four DAPPS (decentralized applications on blockchain). However, their approach does not take into consideration the access control faults and the gas usage of the mutated contracts. The work in [15] developed a mutation testing framework for smart contracts that considers the access control faults, but it does not consider gas. The work in [9] introduced a smart contract mutation approach, but for testing implementations of the Ethereum Virtual Machine (EVM) implementations and not smart contracts. There are two available GitHub repositories with related tools on mutation testing of smart contracts: (1) Eth-mutants[1] which implements just one mutation operator and (2) UniversalMutator which describes a generic mutation tool [13] with set of operators for Solidity.

None of the past works on testing and analysis, focus on automated repair of smart contracts. These works are focused on finding bugs in smart contracts, and not on fixing bugs. Ours is the first proposed approach and tool for smart contract repair.

8.3 Gas usage Calculation of Smart Contracts

The work in [25] presented techniques for calculating the worst case gas usage of smart contracts. Their approach is based on symbolically enumerating all execution paths and unwinding loops up to a certain limit. The authors infer the maximal number of iterations for loops and generates accurate gas bounds. Knowing the worst case gas usage bound for smart contracts can be extremely useful as it provides the smart contract users important information about the maximum amount of gas they need to pay before sending out their transactions to the blockchain networks. The work in [34] provides a graphical user interface that depicts gas usage information (e.g. best and worst case gas usage, and the gas usage of different parts of the code) which helps the developers to optimize the gas usage of their smart contracts.

The past works on gas usage calculation, while relevant to our works, are not directly usable in our repair method. For fast gas usage comparison among patch candidates, we have thus defined and used the notion of gas dominance.

9 DISCUSSION

In this paper, we have presented the first work on automatically repairing smart contracts. Our repair method is gas-aware. The repair algorithm is search-based, and it breaks up the huge search space of candidate patches down into smaller mutually-exclusive spaces that can be processed

[1] https://github.com/federicobond/eth-mutants

independently. The repair technique considers gas usage of vulnerable contracts when generating patches for detected vulnerabilities. Our experiments demonstrated that our method can handle real-world contracts and generate repairs in a short time (less than 1 hour) while taking into consideration the gas consumption of the generated repairs.

Since the owners of smart contracts are unknown, we could not reach out to them in advance, prior to publication. Nevertheless, we hope that our work will spur greater interest in automatically fixing smart contracts via a variety testing, analysis, validation and synthesis methods. We have made our smart contract repair tool and dataset available in GitHub from the following site.

https://SCRepair-APR.github.io

ACKNOWLEDGMENTS

This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by National Research Foundation (NRF) Singapore under National Cybersecurity R&D (NCR) programme, and by a Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (17-C220-SMU-008).

REFERENCES

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. 66–77.

[2] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. 164–186.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. 91–96.

[5] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*. Springer, 849–858.

[6] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography and Data Security - International Workshops, FC 2016, BITCOIN, VOTING, and WAHC, Revised Selected Papers*. 79–94.

[7] Ardit Dika. 2017. *Ethereum Smart Contracts: Security Vulnerabilities and Security Tools*. Master's thesis. Norwegian University of Science and Technology, Department of Computer Science.

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

[9] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 1110–1114. https://doi.org/10.1145/3338906.3341175

[10] Peter Gammie and Ron van der Meyden. 2004. MCK: Model Checking the Logic of Knowledge. In *Computer Aided Verification, 16th International Conference, CAV*.

[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Communications of The ACM* 62, 12 (2019).

[12] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference, POST*. 243–269.

[13] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 25–28.

[14] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL (2017), 48:1–48:28.

[15] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. 2019. Practical Mutation Testing for Smart Contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS International Workshop*. 289–303.

[16] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 259–269.

[17] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

[19] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–379.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[22] Bin Liu, Xiao Liang Yu, Shiping Chen, Xiwei Xu, and Liming Zhu. 2017. Blockchain based data integrity service framework for IoT data. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 468–475.

[23] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 166–178.

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 254–269.

[25] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation*. 450–465.

[26] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.* 20, 1 (2015), 176–205.

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. 448–458.

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 691–701.

[29] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 772–781.

[30] A. Nistor, P. Chang, C. Radoi, and S. Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 902–912. https://doi.org/10.1109/ICSE.2015.100

[31] A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation Testing for the New Century. Chapter Mutation 2000: Uniting the Orthogonal, 34–44.

[32] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* 112 (2019), 275–378.

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. 2014. The strength of random search on automated program repair. In *ACM/IEEE International Conference on Software Engineering*.

[34] Christopher Signer. 2018. *Gas Cost Analysis for Ethereum Smart Contracts*. Master's thesis. ETH Zurich, Department of Computer Science.

[35] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '18)*. 9–16.

[36] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[37] Ron van der Meyden. 2019. On the specification and verification of atomic swap smart contracts. In *IEEE International Conference on Blockchain and Cryptocurrency*. 176–179.

[38] Gavin Wood. 2019. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2019), 1–32.

[39] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. 2019. Mutation Testing for Ethereum Smart Contract. arXiv:cs.SE/1908.03707

[40] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* (Jan. 2017), 34–55.

[41] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 740–751.

[42] Xiao Liang Yu, Xiwei Xu, and Bin Liu. 2017. EthDrive: A Peer-to-Peer Data Storage with Provenance. In *Proceedings of the Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering, CAiSE 2017, Essen, Germany, June 12-16, 2017*. 25–32.