Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Automated Test-Case Generation for Solidity Smart Contracts: the AGSolT Framework and its Evaluation

Stefan W. Driessen*[1] | Dario Di Nucci[1] | Geert Monsieur[2] | Damian A. Tamburri[2] | Willem-Jan van den Heuvel[1]

[1]Jheronimus Academy of Data Science, Tilburg University, Noord-Brabant, the Netherlands

[2]Jheronimus Academy of Data Science, Eindhoven University of Technology, Noord-Brabant, the Netherlands

Correspondence
*Stefan W. Driessen, Sint Janssingel 92, 5211 DA 's-Hertogenbosch Email: s.w.driessen@jads.nl

Present Address
Sint Janssingel 92, 5211 DA 's-Hertogenbosch

Abstract

Blockchain and smart contract technology are novel approaches to data and code management that facilitate trusted computing by allowing for development in a distributed and decentralized manner. Testing smart contracts comes with its own set of challenges which have not yet been fully identified and explored. Although existing tools can identify and discover known vulnerabilities and their interactions on the Ethereum blockchain through random search or symbolic execution, these tools generally do not produce test suites suitable for human oracles. In this paper, we present AGSOLT (Automated Generator of Solidity Test Suites). We demonstrate its efficiency by implementing two search algorithms to automatically generate test suites for stand-alone Solidity smart contracts, taking into account some of the blockchain-specific challenges. To test AGSOLT, we compared a random search algorithm and a genetic algorithm on a set of 36 real-world smart contracts. We found that AGSOLT is capable of achieving high branch coverage with both approaches and even discovered some errors in some of the most popular Solidity smart contracts on Github.

KEYWORDS:
Automated Test Case Generation; Smart Contracts; Blockchain; Search Algorithms; Software Testing.

1 | INTRODUCTION

Blockchain and smart contract technologies are novel approaches to data and code management. They facilitate trusted computing by gracefully allowing for development in a distributed and decentralized manner. Smart Contracts are capsules of code, similar to classes in object-oriented programming languages, such as Java and Python, which are deployed on distributed systems such as blockchains. Smart Contracts and blockchains have seen a major rise in popularity in recent years [1, 2]. In large part, this is due to the inherent qualities of blockchains, such as immutability of data and ease of access to the data stored, which renders extensive testing of critical importance, especially before code deployment. So far, research on testing smart contracts has focused primarily on identifying smart contract- and blockchain vulnerabilities [1, 2, 3, 4], and applying basic techniques such as fuzzing combined with automated oracles to detect these vulnerabilities [5][1]. Automatically detecting vulnerabilities can be a useful tool for smart contract developers, but often having access to a good test suite can prove even more useful to the developer as argued below.

Previous studies have shown that the lack of such test suites is one of the major challenges hampering a successful technology transfer from academics to industry [6, 7, 8]. However, as shown in the extensive investigation conducted by Zou *et al.* [9],

[1]See also Solfuzzer at https://solidity.readthedocs.io/en/develop/contributing.htm#running-the-fuzzer-via-afl

creating test suites for smart contracts is not trivial, and several challenges arise during their implementation. First of all, 54.7% of the interviewed developers report the lack of powerful tools, including testing tools, for blockchain-specific development. Moreover, no mature testing framework and practical testing guidelines are available. Previous tools such as Oyente [2] and ContractFuzzer [5] are undoubtedly promising, but they either do not produce test suites at all or very large test suites, which are not human-readable. Furthermore, these tools do not consider all corner cases and scenarios, which is the most critical challenge raised by the developers interviewed in Zou *et al.* [9]. Finally, they note that currently, no tool is available to measure test suite quality of smart contracts. The only potential exception is represented by the tool developed in Wang *et al.* [10], who follow a similar approach to ours. However, on the one hand, this tool is not publicly available. On the other hand, it does not consider test suite size explicitly, which might result in unnecessarily long and complex test suites.

In this paper, first, we analyze some of the properties for designing a tool for automated test case generation for smart contracts. We believe that these properties are also relevant to researchers who focus on bug detection with automated oracles. We find that properties previously identified [11] for popular programming languages such as Java and C still hold (e.g., deciding on quality metrics and covering corner cases), but additional qualities are desirable, which we describe below. Then, to handle these challenges mentioned above, we present AGSOLT[2] (Automated Generator of Solidity Test Suites), an automated test case generation tool for unit testing for the smart contract programming language Solidity on the Ethereum blockchain. AGSolT creates concise test suites for individual smart contracts while aiming to achieve a high branch coverage level. Existing literature on reducing the size of test suites and test cases can be considered a good initial step for making more human-readable test suites, which are more likely to be used in practice [12, 13]. Therefore, AGSOLT aims at creating smaller test suites, making unit testing[3] and regression testing easier [14, 15]. AGSOLT could lead the way to create higher-quality test suites for Solidity smart contracts that exercise more in-depth corner cases and scenarios combining metaheuristic techniques for the automated test-case generation with developers-provided oracles.

We equip AGSOLT with two common approaches for automated test case generation: (1) fuzzing, which is a random testing approach, and (2) genetic algorithms, which are a search-based testing approach. On the one hand, fuzzing generates test cases randomly; on the other hand, the genetic algorithms iteratively improve an initially random set of test cases through a search guided by one or more objective functions. Previous research [16, 17] has shown that both approaches can be equally effective when generating test suites, which makes them both valid approaches to an automated test case generation problem.

We conducted an empirical study on 36 real-world smart contracts to assess the effectiveness of AGSOLT and take a closer look at how the two approaches compare for Solidity smart contracts. As far as the authors are aware, this is the first comparison of the sort in the domain of smart contracts. We find that AGSOLT achieves good branch coverage on a variety of smart contracts and can detect errors in some of the most popular smart contracts on Github. Both approaches show promise for future investigation, although genetic algorithms might be slightly more suitable for achieving branch coverage on specific types of smart contracts. Although neither approach is significantly faster than the other, our experiments seem to indicate that a guided search that prefers smaller test cases might be better at reducing the time spent running the tests on a blockchain implementation.

In sum, this paper contributes to the state-of-the-art by:

1. Proposing a set of challenges specific to the blockchain domain that any automated test case generation tool should aim to overcome.

2. Introducing AGSOLT: an automated test case generation tool, capable of: (i) creating small, human-readable test suites that are optimized for branch coverage; (ii) allowing for the implementation of different types of algorithms such as random-testing and search-based-testing; (iii) being easily adapted to allow for different types of objectives such as mutation coverage or statement coverage.

3. Providing the first comparison between a guided search and a random search in the domain of automated test case generation for smart contracts.

The rest of this paper is organized as follows: Section 2 introduces the concept of smart contracts in the context of the Ethereum blockchain and discusses existing ATG tools for these smart contracts. Section 3 formalizes the challenges that we identify for creating an ATG tool for Smart contracts. In Section 4, the AGSOLT tool is introduced, and its workings are explained. Section 5

[2] https://github.com/AGSolT/AGSolT2021Submission
[3] In the domain of smart contracts, *unit testing* means testing individual smart contracts.

describes the design and the results of the empirical study we conducted to evaluate AGSOLT and compare the search-based and random algorithms, while Section 6 discusses its threats to validity. Finally, Section 7 discusses the results of these experiments and introduces potential future work.

2 | BACKGROUND

This section provides an overview concerning blockchain, smart contracts, and their testing.

2.1 | The Ethereum Blockchain and Smart Contracts

A blockchain [18, 19] can be viewed as a *decentralized*, *distributed* digital ledger: an ordered list of *blocks*, which themselves contain an ordered list of *transactions*. New blocks are added by *miners*, who follow a *consensus protocol* that dictates the rules of the blockchain, including how to add new data and deal with conflicting versions of the blockchain. On the Ethereum blockchain, transactions can be used to transfer *Ether* (ETH) cryptocurrency from one address to another and deploy and interact with *smart contracts*. Ether is also used to compensate the miner, who receives a small fee (called *Gas*) from the transaction sender for registering a transaction on the blockchain. In addition to sending Ether, transactions can also be used to deploy- and interact with smart contracts Because of its inner working, the Ethereum blockchain can store (almost) any data type, so long as modifications are made in a transaction-based manner. Its creators have leveraged this property to store (compiled) pieces of code, called smart contracts, on the blockchain, which can be used as follows. Each transaction has a "Data" field where a transaction sender can store bytecode. When sending to a previously unused *address*, this bytecode can be interpreted by miners that use the *Ethereum Virtual Machine* (EVM) to create new smart contracts whose bytecode is stored on the blockchain at the new address.

After a contract has been deployed, the transactions sent to its address can invoke the execution of the code stored on the blockchain by including the method to be invoked and any input parameters in the Data field of the transaction. The EVM specifies how to alter the state of the system based on the Data field of the transaction and the code stored at the specified address [20]. If a transaction is issued without a recognizable method in its "Data" field, a special function called FALLBACK is invoked.

Since writing bytecode by hand is impractical, several high-level programming languages have been created, the most popular of which is *Solidity*, which is inspired by Python, C++, and Javascript [4]. Smart contracts in Solidity are similar to *classes* in object-oriented programming and behave similarly to *objects*: the smart contract code serves as a blueprint to deploy many instances on the blockchain, each with their address and internal state. Similarly, Solidity smart contracts have both public functions and variables that can be accessed from outside the smart contract and private functions and variables that can only be interacted with by the contract itself.

2.2 | Smart Contract Weaknesses and Testing

Detecting vulnerabilities in smart contracts has been a hot research topic in recent years, especially since the infamous DAO (Distributed Autonomous Organisation) attack in 2016, where roughly 60 million dollars worth of Ether was stolen because of an unforeseen exploit in a published smart contract [21]. Due to specific blockchain properties, such as the immutability of committed blocks and its distributed and decentralized nature[5], the proper implementation of smart contracts is particularly challenging. We discuss the most relevant literature below to illustrate this point.

Delmolino *et al.* [3] found that when teaching undergraduate students to create smart contracts, even simple implementations lead to a multitude of non-trivial problems. Often, such problems do not prevent compilation but leave the contract vulnerable to exploitation or unintended behaviors. Anderson *et al.* [1], Luu *et al.* [2], and Atzei *et al.* [4] investigated already published contracts and highlighted that some of them present design flaws although already published on the blockchain. Recently, Zou *et al.* [22] investigated the challenges related to smart contract testing and confirmed that almost half of all developers desired

[4] https://solidity.readthedocs.io/en/v0.5.8/

[5] Distributed in this context implies that anyone can access the bytecode of a smart contract, decentralized means that anyone can interact with a deployed contract.

tools to verify code correctness. The above studies and the previously mentioned DAO attack motivated introducing new development tools to develop and test safe smart contracts effectively[6]. Several tools have been put forward that we introduce briefly below: SOLIDITY-COVERAGE [7] measures the quality of an existing test suite by checking whether *branch coverage* [23] has been achieved (i.e., whether all possible paths through the code have been executed). SOLIDITYCHECK [24] checks Solidity code for patterns that are known to lead to vulnerabilities and warns the user about them. Wu *et al.* [25] have designed 15 mutation operators for Solidity smart contracts and use these to detect defects in 26 real-world smart contracts. OYENTE [2] creates a control-flow graph for a given smart contract and uses symbolic execution to check its *branch feasibility*, (i.e., whether each part of the code is theoretically reachable), as well as whether vulnerabilities are present. ADF-GA [26] uses control-flow-graphs with dup-based covering criteria but only tests on a small set of smart contracts that only use integers and unsigned integers. Similarly, Wang *et al.* [10] propose a tool propose a tool for creating branch-covering test suites that they test on 8 smart contracts. A recent addition by Liu et. al. is MODCON, which relies on user-defined models to impose model testing on smart contract [27]. Finally, *fuzzers* [28] automatically create test cases for smart contracts by generating random (within a specified range) inputs for contract functions to detect errors. The commercial ECHIDNA [29] tries to break user-defined invariants, while the academic CONTRACTFUZZER [5] checks for both coding errors and the vulnerabilities mentioned by Luu *et al.* [2] and Bartoletti *et al.* [30].

When it comes to automated unit-testing of Solidity smart contracts on the Ethereum blockchain, each of the approaches mentioned above comes with its limitations: (i) SOLIDITY-COVERAGE[7], SOLIDITY CHECK [24], and OYENTE [2] do not produce test suites; (ii) ECHIDNA [29] and MODCON [27] require the user to define invariants or models of their code, and (iii) ADF-GA [26] and tool of Wang *et al.*are tested on small subsets of possible smart contracts and do not provide online code to be used for further research. CONTRACTFUZZER [5] is perhaps the most complete approach out there because it creates test suites fully automatically and works on a variety of smart contracts. However, the tool focuses on vulnerability detection through automated oracles as opposed to creating *test suites* which can be used by human oracles. Additionally, existing literature has suggested that random search approaches (e.g., fuzzing) run the serious risk of being too simplistic to fully capture corner cases in more complex applications when compared to guided search approaches [17, 31].

For these reasons, we introduce AGSOLT (Automated Generation of Solidity Test Suites). This tool can easily leverage different search algorithms to automatically generate test suites for Solidity smart contracts that aim to achieve branch coverage. In the next sections, we first introduce the challenges that *any* tool or framework that sets out to achieve this goal will meet and then discuss how AGSOLT aims to overcome these challenges. Finally, we demonstrate the effectiveness and efficiency of the tool by experimenting on 36 real-world smart contracts.

3 | SMART CONTRACT TESTING

This section introduces a set of properties that an effective automated test suite generation tool should possess. These properties were found through iterative experimentation, aiming at finding the corner cases not covered by the two search strategies exploited by AGSOLT. In particular, we started implementing existing algorithms and looking at the branches that these were unable to cover and identified the causes. We describe here, those blockchain-specific properties we found, which have not yet been identified in the literature. We direct the reader towards existing literature [32, 33, 34, 35, 36] for more background on the general challenges of test case generation, such as choosing objectives, objective functions and improving efficiency and effectiveness. We divide these properties into three different types: *transactional properties* of blockchains and smart contracts, *properties of the blockchain* on which the smart contract is deployed, and properties that define the way smart contracts *interact* with other smart contracts. We argue that any tool for automated test-case generation should consider these properties.

3.1 | Transaction Properties

The only way to change the state of a smart contract is by sending a *transaction* to the contracts' address and invoke one of its functions. Besides the function and parameter specification, *every* interaction with a smart contract has to provide a sender, which is the address from which the transaction was sent, a value[8], which is the amount of Ether sent in the transaction, and an amount of gas, which is the fee that the sender has to pay to the miner for the computational power involved in adding

[6] https://github.com/ethereum/wiki/wiki/Safety#ethereum-contract-security-techniques-and-tips
[7] https://blog.colony.io/code-coverage-for-solidity-eecfa88668c2/
[8] Many blockchain interaction platforms do not require a value be specified, in which case this defaults to zero.

this transaction to the block. These *transaction properties* can be accessed by the smart contract receiving the transaction and influence its inner workings, affecting which branches are traversed.

```
1  pragma solidity 0.5.12;
2
3  contract Auction {
4    address payable public Seller;
5    address payable public Frontrunner;
6    uint public HighBid;
7    uint public CloseTime;
8
9    constructor(uint _CloseTime) payable public {
10     Seller = msg.sender;
11     Frontrunner = msg.sender;
12     HighBid = msg.value;
13     CloseTime = _CloseTime;
14   }
15
16   function Bid() payable external{
17     require(msg.value > HighBid);
18     Frontrunner.transfer(HighBid);
19     HighBid = msg.value;
20     Frontrunner = msg.sender;
21   }
22
23   function Claim() external{
24     require(block.timestamp > CloseTime);
25     // Implement ownership transfer
26     selfdestruct(Seller);
27   }}
```

Smart Contract 1: An example of Ethereum-specific properties.

As an illustration, Smart Contract 1 shows an example of a simple Auction on the Ethereum blockchain. When the contract is initiated the constructor is executed, which instantiates the SELLER, FRONTRUNNER, HIGHBID and CLOSETIME variables. Afterwards anyone can make a Bid by calling the BID() function (lines 16-21). This function first checks if the transaction property MSG.VALUE (the new bid) is higher than the current highest bid and if it is, it refunds the previous highest bidder (FRONTRUNNER) and changes the Highest Bid and frontrunner based on the transaction information MSG.VALUE and MSG.SENDER respectively.

Any automated test-case generation tool for smart contracts should generate test-cases containing transactions from different accounts to test sender-dependent functionality. Similarly, the tool should vary the amount of Ether send with a transaction and evolve it either as if they were input variables or chosen for this purpose.

3.2 | Blockchain Properties

Besides transaction properties, a smart contract has access to additional information from the blockchain environment on which it is deployed, such as the address of the miner of the current block, the gas limit of the current block (i.e., the maximum amount of computation in a block), the hash of any of the least 256 recently added blocks, and the time and block number of the current block. Moreover, because each smart contract has an address, it has a balance in Ether associated with it, which affects its ability to send Ether. An example of this is given by the CLAIM() function in Smart Contract 1 which compares the blockchain property BLOCK.TIMESTAMP (which gives the time since the Unix epoch for this block) with the user-specified CLOSETIME before the auction can be closed. If the specified time has been reached, the smart contract removes itself from the blockchain and sends its entire balance to the seller. These blockchain properties can be manipulated (within certain limitations) by the test environments. A useful test case generation tool should vary some or all of these blockchain properties for better testing while at the same time respecting the logical rules of the blockchain, such as that block numbers and time must always increase between different blocks.

3.3 | Interactive Properties

Similarly to how Java classes can instantiate and interact with other classes, smart contracts on the Ethereum blockchain can instantiate and send transactions, such as method invocations or Ether transfers, to other smart contracts. However, there are two essential differences. First, smart contracts can send transactions to any address on the blockchain, allowing them to transfer

Ether to a wallet or call functions of *any* smart contract on the same blockchain as long as that contracts address is passed as a variable to the calling smart contract. A special case occurs when the contract sends a transaction to the so-called *zero-address* (0x0), which causes the instantiation of a new smart contract. Second, smart contracts have an Ether balance that they can use to send Ether alongside transactions to other smart contracts to invoke the contract's functionality or purely transfer currency.

At line 18 of the BID() function in Smart Contract 1 the previous FRONTRUNNER is refunded the bid. If such FRONTRUNNER is a smart contract, this transaction invokes the fallback function of that smart contract, which could, in turn, call one of the functions in the AUCTION smart contract.

Automated Test Case Generation Tools for Smart Contracts should be aware of both existing addresses, as well as non-existing addresses and the zero-address, which might cause errors in the smart contract. Additionally, they should anticipate interaction with smart contracts outside the programmer's control.

4 | AGSOLT: AUTOMATED GENERATOR OF SOLIDITY TEST SUITES

This section describes the design choices and algorithmic procedures that make up the main workings of AGSOLT. Figure 1 shows its high-level workings, which is composed of an *initialization phase* and a *testing loop*. In the initialization phase, relevant properties of the smart contract(s) under examination are extracted, which are required during the testing loop. During the testing loop, test cases are run on a blockchain implementation[9] and their performance is evaluated using the branch distance. AGSOLT is mostly implemented in Python, except the instrumentation of the blockchain, which is done through the WEB3[10] library in Javascript. Our ultimate goal is to achieve branch coverage. This concept can be intuitively understood by viewing a piece of code as a graph with nodes that contain grouped statements and edges that indicate jumps between these groups brought about by conditions (such as if-else statements) that control which groups of statements are executed and which are not. Sections 4.1.2 and 4.1.3 describe how AGSOLT creates these graphs and gives some examples for further illustration. In sections 4.2.1 and 4.2.2 we demonstrate the two implemented approaches for creating test suites whose test cases traverse all (or as many as possible) of these edges. AGSOLT considers each branch to be covered, as a separate objective and tries to improve upon each objective simultaneously, thus offering multi-objective optimization.

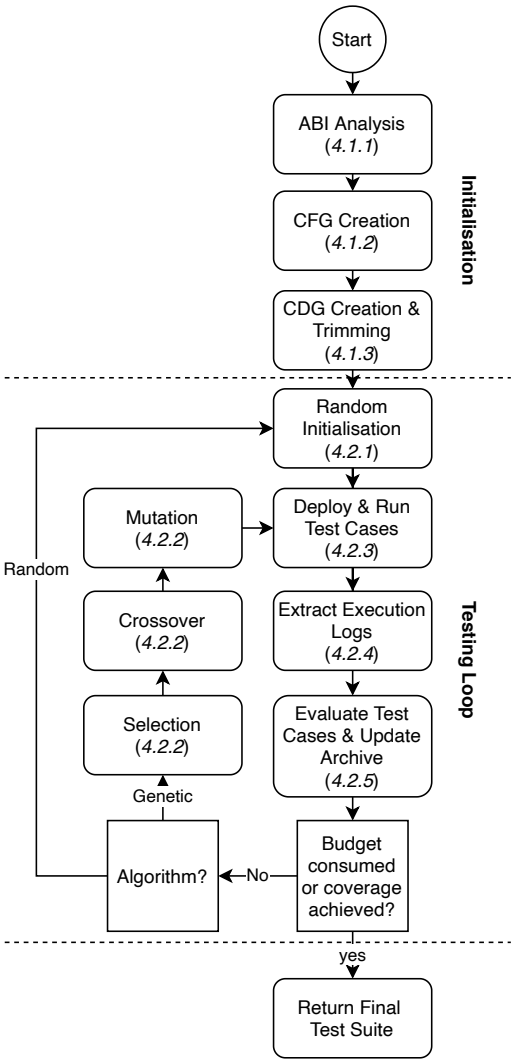

FIGURE 1 The flowchart of AGSOLT; each step is explained in a corresponding section.

4.1 | Initialization Phase

During the initialization phase, AGSOLT extracts several characteristics of the smart contract under investigation to create the first generation of test cases that can be improved in the *testing loop*.

4.1.1 | ABI Analysis

When Solidity code is compiled into bytecode, an APPLICATION BINARY INTERFACE (ABI) file is created. This file contains the basic information necessary for test case generation (i.e., function names and input types). Additionally, during this step, hard-coded values of the contract are scraped to be used for *seeding* the method invocations. Seeding is a common technique used in automated test case generation, which involves including

[9] https://www.trufflesuite.com/ganache
[10] https://web3js.readthedocs.io

certain values with higher probability when randomly selecting variables for input variables [37]. In AGSOLT whenever a random input variable or ETH value or account is selected, first a check is performed whether one or more hard-coded values of the corresponding type were present in the smart contract (and consequently scraped). If there are such values, 50% of the time a random scraped value is picked instead of a completely random value. This allows AGSOLT to automatically leverage information inside the smart contract to reach branch coverage quicker.

TABLE 1 EVM opcodes needed to generate test cases

Hex	Opcode	Stack Input	Stack Ouput
56	JUMP	dest	
57	JUMPI	dest, bool	
5B	JUMPDEST		
10	LT	a, b	$a < b$
11	GT	a, b	$a > b$
12	SLT	a, b	$a < b$
13	SGT	a, b	$a > b$
14	EQ	a, b	$a = b$
15	ISZERO	a	$a = 0$

4.1.2 | Control Flow Graph Extraction

To keep track of the branches to be traversed and those already covered, EVOSOL extracts the *control dependency graph* [38] of the smart contract. To this end, first the *Control Flow Graph* (CFG) is distilled from the bytecode using the python EVM_CFG_BUILDER library[11]. A conceptual explanation of how a CFG can be extracted from bytecode is given in our online appendix[12]. The reason the CFG is created from the bytecode (as opposed to the Solidity code), is because it makes it possible to extract the values that are on the stack when the EVM evaluates a predicate controlling a branch. These values are needed later for deciding how close a test case is to satisfying a predicate, and consequently, traversing the branch it controls. Table 1 shows the 9 opcodes which are relevant for identifying nodes and branches in the opcode column, their hex value as it appears in bytecode as well as the argument(s) they consume from the stack and the output value they push onto the stack. The "JUMP"-opcode is used to jump to a different part of the bytecode for execution (indicated by the destination value). The "JUMPI"-opcode works similar to "JUMP" except that execution continues from the destination, only if the consumed bool is true, this is what creates branches in the CFG. Finally the other opcodes shown in Table 1 correspond to the predicates that can control a branch; $<$, $>$, $==$ and $\neg$. Note that $\leq$, $\geq$ and $\neq$ can be represented with $\neg >$, $\neg <$ and $\neg ==$ respectively. For each branching node, AGSOLT identifies the opcode that corresponds to the controlling predicate to compute the branch distance for the outgoing branches.

4.1.3 | Control Dependency Graph Creation and Optimization

Some edges of the graph lead to superfluous nodes whose execution neither leads to or depends on any predicate, that could waste part of the search budget. Therefore, they are eliminated by running the COMPACTIFYCFG algorithm shown in Algorithm 1 which uses the COMPACTIFY procedure in Algorithm 2. Finally, AGSOLT uses the algorithm proposed by Lengauer and Tarjan [39] to determine the control dependencies between the nodes and distil the *Control Dependency Graph* from the *Control Flow Graph*. Considering some specific characteristics of the Ethereum blockchain, the graph can still be optimized by removing some nodes that are not relevant for test case generation. These nodes and edges belong to the following patterns:

[11] https://github.com/crytic/evm_cfg_builder
[12] https://github.com/AGSolT/AGSolT2021Submission/tree/master/CFG_Creation_Appendix

Algorithm 1 COMPACTIFYCFG

Input:
N ▷ The set of all nodes in the CFG.
Output:
N' ▷ The set of nodes where nodes with superfluous branches have been merged.

procedure COMPACTIFYCFG
  $UN \leftarrow N$ ▷ Initialise the unmerged nodes.
  $MN \leftarrow \emptyset$ ▷ Initialise the merged nodes.
  while $UN \neq \emptyset$ **do**
    $Node \leftarrow any\ n \in UN \mid n.incoming\_nodes \cap UN = \emptyset$
    $UN \leftarrow UN - Node$
    $MN \leftarrow MN \cup \{Node\}$
    $UN, MN \leftarrow Compacti\,f\,y(Node, UN, MN)$
  end while
return MN
end procedure

Algorithm 2 COMPACTIFY

Input:
$Node$ ▷ A node to be compactified.
UN ▷ The set of unmerged nodes.
MN ▷ The set of merged nodes.
Output:
UN' ▷ The updated set of unmerged nodes.
MN' ▷ The updated set of merged nodes.

procedure COMPACTIFY
  if $\#Node.outgoing\_nodes \neq 1$ **then return** UN, MN
  else if $\#Node.incoming\_nodes \neq 1$ **then return** UN, MN
  else
    $nextNode \leftarrow Node.outgoing\_node$
    $UN \leftarrow UN - nextNode$
    $MN \leftarrow MN \cup \{nextNode\}$
    $Node \leftarrow Node \bigoplus nextNode$
  end if
return COMPACTIFY($Node$, UN, MN)
end procedure

- **Dispatcher Nodes and Edges.** The ETHEREUM bytecode contains a dispatcher function that handles the transactions to the smart contract. Since AGSOLT invokes all (public) methods, there is no need to calculate the branch distance for these edges.

- **Empty Fallback.** An empty fallback function is initialized when the user does not explicitly define one. However, such a function can be safely ignored as it does not change the semantics.

- **State Variables.** Public variables are accessed as functions through the contract dispatcher. Since calling these variables does not help cover new branches, the corresponding nodes and edges in the CDG can be ignored.

An example of these patterns is shown in Fig. 2: The control dependency graph of smart contract 1 starts with dispatcher nodes (even nodes), which are used to identify the method or state variable (starting at uneven nodes) that is called. If none of the state variables or methods was passed in the transaction, the fallback function (starting at node 12) is invoked. Since no fallback function was specified in smart contract 1, this method is empty, and neither invoking it nor any of the state variables is particularly interesting for testing purposes. For that reason AGSOLT, removes the edges and nodes corresponding to the dispatcher, state variables and empty fallback functions (shown dotted in Fig. 2 and creates new edges to the relevant methods (BID and CLAIM) which are shown in bold in Fig. 2.

- **Payable Check.** A SOLIDITY function can be declared as *payable* if it accepts transactions that have an associated ETHER value. When a function is not declared as payable, the Ethereum compiler makes sure that the EVM reverts such transactions. Since our goal is to test only the functionalities implemented by the developer, AGSOLT ignores such branches and simply does not send ETHER to non-payable functions.

As an example, Figure 3 shows the CDG of the `Claim` function reported in Smart Contract 1. Before going from line 23 to 24, the EVM verifies if the transaction has an ETHER value and, if it does, reverts the transaction. AGSOLT trims the dashed nodes and edges and merges the start node with node 3.

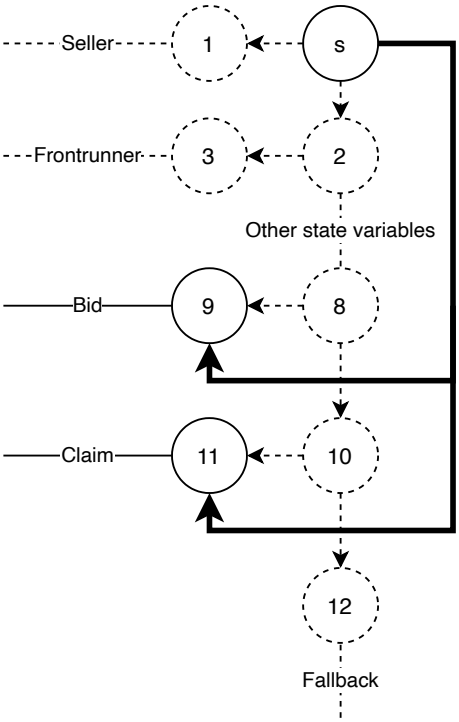

FIGURE 2 CDG of the `dispatcher` function in Smart Contract. 1

4.2 | Testing Loop

During the testing loop, the actual search for optimal test cases is performed until the budget is consumed. We first discuss the difference between a random- and a search-based algorithm, followed by the general steps.

4.2.1 | Random Initialisation of Test Cases

After extracting the required information, the population of test cases is initialized through a random algorithm. As in previous work, each test case is a sequence of statements $t = \langle s_1, s_2, ..., s_n \rangle$ [13, 40, 41]. AGSOLT relies on two types of statements:

- **Constructor statements** are used to deploy smart contracts on the blockchain. Such statements are used as the first statement s_1 of each test case t to ensure that a fresh instance of the smart contract is instantiated for each test case on which the function statements can be called. This statement type contains the information required to deploy an instance of the relevant smart contract on the blockchain, including the input variables required by the smart contract constructor and the transaction metadata, such as the amount of ETH send with the transaction and the account from which the transaction is sent.

- **Function statements** are used to create transactions that invoke functions in the deployed smart contracts. Indeed, the only way to interact with a smart contract in Ethereum is by sending a transaction to its address. All the statements, but the first (i.e., the constructor statement), in a test case are function statements that are responsible for traversing the branches of the smart contract. This statement type contains a reference to the function to cover, its input variables, and the transaction metadata.

A set of test cases is initialized by creating N random test cases, where N is the population size i.e., the number of test cases in any generation. When AGSOLT relies on the random search, test cases are generated by performing only this step. The search keeps running through until either full branch coverage is achieved or the specified *budget* is consumed. At this point, the final population (i.e., the archive) is presented as the solution. As shown in Figure 1, to improve the generated test cases, AGSOLT can perform a guided search and a random search. For the former, we integrated DYNAMOSA (i.e., Many-Objective Sorting Algorithm with Dynamic target selection), the genetic algorithm proposed by Panichella *et al.* [12].

4.2.2 | Genetic Loop: the DynaMOSA Algorithm

Genetic Algorithms are inspired by biological evolution: they work with a population of (candidate) solutions or *chromosomes* from which they derive a next *generation* of solutions by iteratively applying *evaluation*, *selection*, *crossover*, and *mutation*. Mitchell [32] and Lucken *et al.* [42] provide more details on genetic algorithms for multi-objective problems. DynaMOSA [12] is a state-of-the-art algorithm specifically designed for automated test case generation. It facilitates the creation of a small and effective test suite through multi-objective optimization inspired by NSGA-II [34]. DynaMOSA has been shown to significantly outperform other test case generation algorithms (e.g., Whole-Suite Approach [13] and LIPS [35, 43]) in terms of branch and mutation coverage on an extensive set of Java classes.

Fitness Function. The search algorithm is guided by the normalized branch distance, as defined by Arcuri *et al.* [13]. The normalized branch distance for a test case t and a branch b with controlling predicate p_b is given by:

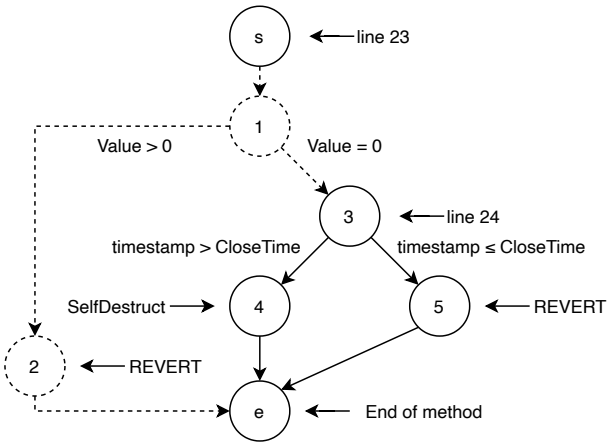

FIGURE 3 CDG of the `Claim` function in Smart Contract 1

$$d(t,b) = \begin{cases} 0 & \text{if } t \text{ satisfies } p_b, \\ \frac{f_{p_b}(t,b)}{f_{p_b}(t,b)+1} & \text{if } p_b \text{ has been reached but not satisfied,} \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

Here $f_p(t, b)$ is given by Korel's objective function for relational predicates as shown in Table 2 [44]. Test cases with a smaller normalized branch distance are closer to covering the corresponding branch and are thus more desirable.

TABLE 2 Relational predicates and objective functions [44]

Relational predicate	f_p
$a > b$	$b - a$
$a \geq b$	$b - a$
$a < b$	$a - b$
$a \leq b$	$a - b$
$a = b$	$abs(a - b)$
$a \neq b$	$-abs(a - b)$

Because we aim at covering all branches simultaneously, the goal of the search becomes the following, similarly to what previously formulated by Panichella *et al.* [12]:

Definition 1 (Fitness Function)**.** Let $B = \{b_1, b_2, ..., b_k\}$ be the set of branches in a smart contract. Find a test suite $T = \{t_1, t_2, ..., t_n\}$ consisting of non-dominated test cases t that simultaneously minimizes the fitness function for each branch $b \in B$, i.e., minimizing the following k objective functions:

$$\begin{cases} f_1(t) & = al(t, b_1) + d(t, b_1) \\ f_2(t) & = al(t, b_2) + d(t, b_2) \\ \vdots & \\ f_k(t) & = al(t, b_k) + d(t, b_k) \end{cases} \tag{2}$$

Here $al(t, b_i)$ is the *approach level* of t to b_i (i.e., the number of predicates between the closest branch executed by t and b_j) and $d(b, t)$ is the minimal *normalized branch* distance of t to branch $b \in B$ as defined in Equation (1).

Note that in this multi-objective approach a distance is calculated for each objective (branch) so that rather than using a *single* distance to describe the fitness of a test case t, a distance *vector*, $\vec{d_t} = \langle d(t, b_1), d(t, b_2), ..., d(t, b_n)\rangle$, is used.

Selection Operation. After randomly initializing the first generation of test cases and measuring the branch distances, the test cases are ranked using their Pareto fronts [34] as the primary criterion. Although in multi-objective optimization having solutions that make trade-offs between objectives is usually desirable, this is not the case for automated test case generation. Indeed, only fully covered branches are relevant for the branch coverage, whereas a test that *almost* covers one (or more) uncovered branches does not add any value to the final test suite. When ranking test cases in the first Pareto front, DYNAMOSA uses a preference criterion that generalizes this idea by determining, for each branch $b \in B$, the non-dominated test cases closest to covering b and (if there are more than one) the shortest one among those. More formally, as defined by Panichella *et al.* [12], the preference criterion is the following:

Definition 2 (Preference Criterion.)**.** Given a branch b_i with corresponding objective function $d_i = d(b_i, t)$, a test case t is preferred over another test case t' (written as $t \prec_{b_i} t'$) *iff*

$$d_i(t) < d_i(t') \text{ OR } d_i(t) = d_i(t') \wedge size(t) < size(t'). \tag{3}$$

where $size$ is a function that gives the length (e.g., number of statements) of a given test case. Size is considered a secondary criterion to prioritize solutions because shorter solutions reduce the oracle cost for humans [45, 13].

To compare the test cases that are not in the same Pareto front and are not preferred by the preference-criterion, the *sub-vector-distance-assignment* algorithm introduced by Köppen and Yoshida [46] is used as a secondary selection criterion. Its goal is selecting the most diverse possible subset of solutions from the last Pareto front for the next generation.

Crossover and Mutation Operations. We apply crossover and mutation on a test case level, following the approach suggested by Arcuri *et al.* [47]. First, two parent test cases p_1 and p_2 are selected from the previous generation using tournament selection [34]. Each parent is then cut into two parts, and the first part from p_1 is combined with the second part of p_2 and vice versa two create two child test cases. Mutation is performed by randomly applying *remove, change* and *insert* operators. These operators remove statements, slightly change the variables in the statements or insert new statements into test cases, respectively.

4.2.3 | Deploy & Run Test Cases

Before evaluating and selecting the best test cases for the next generation, each test case runs on an Ethereum blockchain environment. As mentioned in Section 4.2.1, each test case starts with a constructor statement, which is used to deploy a new instance of the smart contract to the blockchain instance. By looking at the receipt of the transaction, AGSOLT instantiates the new smart contract and extracts its address on the blockchain. Afterward, each method call is executed by sending a transaction to the instance's address. The hash-codes of the transactions, which identify each transaction on the blockchain, are stored for the next step.

4.2.4 | Extract Execution Logs

To compute the branch coverage for a test case as defined in Equation (1), two types of information are required: (i) the parts of the code covered by the test case and (ii) the values that are on the stack when a branch-controlling predicate is evaluated. AGSOLT extracts this information through a slightly modified functionality of the javascript WEB3[10] debug module called *getTransactionTrace*. This module takes the transaction of a method call, recreates the blockchain state when the transaction was executed, and writes the executed opcodes and the stack evolution in a file used for the next evaluations.

4.2.5 | Evaluate Test Cases & Update Archive

After executing all the test cases and retrieving the necessary information, test cases are evaluated, as shown in Algorithm 3, to produce the distance vector, *test_scores*, describing the test case's fitness. For each test case, its distance to all branches is initialized as infinite (line 2). Additionally, AGSOLT keeps track of all traversed edges (initialized at line 3) to calculate the approach levels for those edges whose starting nodes are not reached during the execution. For every method call in the test case, AGSOLT takes the corresponding list of executed opcodes and a list of lists containing all the values on the stack when executing each opcode (line 4). The first node in the CDG of any method is always the same (line 5), while its end is only reached when a node has no outgoing edges (line 6). Finding the next node using the FINDNEXTNODE (line 7) method means looking at the first opcodes executed after leaving the current node and comparing them to the opcodes of the nodes with an

Algorithm 3 Evaluate Test Case

Input:
$methodCalls$ ▷ The list of methods called by the test case
$Opcodelists$ ▷ The lists of opcodes executed by each Methodcall
$Callstacklists$ ▷ The lists of all the items on the stack when each opcode was executed
$Edges = \{E_1, E_2, ..., E_n\}$ ▷ The (ordered) list of Edges of the smart contract.
$Nodes = \{N_1, N_2, ..., N_m\}$ ▷ The (ordered) list of Nodes of the smart contract.
Result: a distance vector which contains the test case's distance to each branch.

```
    procedure SET DISTANCES
        test_scores = [∞, ∞, ..., ∞]        ▷ Distance to each Edge
        traversed = ∅        ▷ The set of traversed edges
        for each Methodcall, Opcodelist, Callstacklist do
 5:         curNode = startNode
            while curNode ≠ endNode do
                nextNode = FINDNEXTNODE(curNode, Opcodelist)
                for each E_i ∈ Edges do
                    if E_i.startNode == curNode then
10:                     test_scores[i] = min(test_scores[i],
                        BRANCHDIST(Opcodelist, Callstacklist, E_i))
                    end if
                    if E_i.endNode == nextNode then
                        traversed = traversed ⋃ {E_i}
15:                 end if
                end for
                curNode = nextNode
            end while
        end for
20:     for each E_i ∈ Edges do
            if test_scores[i] == ∞ then
                test_scores[i] = APPROACHLEVEL(E_i, traversed)
            end if
        end for
25: end procedure
```

incoming edge from the current node. For each reached node, AGSOLT analyzes the outgoing edges (lines 8-9) and updates the *test_scores* if the normalized branch distance from Equation (1) is smaller than the smallest distance found so far in the test case (lines 10-11). After identifying all traversed edges, AGSOLT calculates for each not covered branch, the approach level: i.e., the number of edges that would need to be traversed before the node controlling the branch can be reached (lines 19-23). Finally, if a test case outperforms the best test-case found so far for a particular branch, it is stored in an *archive*, which keeps track of the best test-case for each branch. It is important to note that (as can be seen in Fig. 1) both the random testing approach and the genetic approach go through steps 4.2.3 through 4.2.5. The key difference between these approaches is that genetic algorithms use *selection*, *crossover*, and *mutation* to create the next generation of test cases, while random testing creates a new set of randomly initialized test cases.

4.3 | Dealing with Blockchain Properties

To deal with the challenges that arise from transaction properties, blockchain properties, and interactive properties mentioned in Section 3, AGSOLT provides configuration options that deal with the Ethereum and Solidity blockchain and smart contract environment:

- **Transaction Properties.** AGSOLT extracts all the accounts of the blockchain environment and uses them both as senders of transactions and as input variables whenever an address type is required. It keeps track of whether a function is payable and, if so, it sends an amount (between a configurable maximum and minimum) of Ether with the transaction. Both addresses and values can be evolved by the genetic algorithm as though they were input variables.
- **Blockchain Properties.** AGSOLT allows the user to include a `PassBlocks` or `PassTime` method call in test cases, which instruct the blockchain environment to update the latest block number or the time rather than invoke smart contract functions (assuming the chosen blockchain environment allows these manipulations). Both block number and time can only *increase*, similarly to real-world Ethereum implementations. The miner configurations can be set in the blockchain environment. Therefore, they are not manipulated in AGSOLT.

- **Interactive Properties.** In addition to using the extracted accounts as input variables, AGSOLT has an option to include specific non-existent accounts, which can trigger specific errors. This feature also allows the users to indicate a new contract creation through the zero-address (0X0). Smart contracts can be deployed in the blockchain environment before the test suite generation. Their address can be provided to AGSOLT as address input variables to test the interaction between the contracts.This offers a simple, yet effective way, for users to create e.g., stubs with their own desired functionality and test interaction properties. Additionally, the user can use this functionality to provide addresses that do not exist on the blockchain as input variables for the contract functions to test the behavior of the contracts when the sent transactions fail.

4.4 | Resulting Test Suites

At the end of the procedure shown in Fig. 1 AGSOLT outputs a text-file that gives information about the test suite and the test process. In particular, it includes the number of branches found and covered, the number of iterations through the loop before stopping, the total time spent testing, and the time spent running the tests on the blockchain. Afterward, the test cases are provided as construct statements and method calls with relevant input- and transaction arguments. The test suite is easily interpretable for humans and can easily be automatically transformed into input for the user's preferred testing environment.

In addition to the test suite, AGSOLT writes out the same meta information of *all* contracts that were tested in a CSV file for easy comparison.

5 | EMPIRICAL EVALUATION

This section reports the empirical study that we performed to compare *effectiveness*, *efficiency*, and *test case length* of the two algorithms for test case generation implemented in AGSOLT: namely, a *fuzzer* and *DynaMOSA* [12].

5.1 | Data Collection

In an attempt to test on real-world smart contracts for our experiment, we scraped Github to obtain the most starred projects containing Solidity files. We selected the smart contracts that adhered to the following criteria: (i) being stand-alone, meaning they do not call other smart contracts during run-time (although they can inherit functionality from other smart contracts), (ii) coming from different application domains, (iii) not having any user-defined inputs for their functions. We retrieved 36 Solidity smart contracts from 17 different repositories, which is comparable to existing studies [26, 27]. To confirm that the contracts were used in the real world, we manually inspected them, and we found that at least 17 of the smart contracts have also been deployed on either the main Ethereum network or on a test network. Table 3 shows the characteristics of the identified smart contracts, including their domain, whether they were found online, their number of statements, and number of branches in its CDG. Additionally, Table 3 highlights presence the blockchain-specific qualities that AGSOLT can handle. The *sender dependence* and *value dependence* indicate whether functionality of the smart contract depends on the transaction sender and transaction value and fall into the transaction properties discussed in Section 3 and Section 4.3. *Block dependence* and *time dependence* indicate whether the contract relies on block number or the blockchain time for its functionality, which falls into the blockchain properties discussed in Section 3 and Section 4.3. Finally *account as variables*, *non-existing account dependence* and *zero account dependence* indicate the presence of interaction within the smart contract that would depend on the accounts passed as input variables and fall into the interaction properties discussed in Section 3 and Section 4.3. The entire data set, including the addresses of the deployed smart contracts, along with the tool and the results, is available in our online appendix[2]. The smart contracts are spread out over ten application domains. They vary in terms of the number of source code statements and branches in the CDG of the corresponding bytecode. We found that the transaction properties we identified occurred most frequently (28 sender dependencies and 26 value dependencies), followed by the interaction properties (29 variable dependencies, four non-existent account dependencies, and two zero-account dependencies). Interestingly only three smart contracts exhibited blockchain properties (two time dependencies and one block dependency). This characteristic is feasible because relying on block and time information is inconsistent (each miner might have different information), and developers should rely on it as little as possible. Importantly only four smart contracts do not rely on any of the properties we identified. Since the presence of these dependencies was not part of the search protocol, this demonstrates the necessity for our tool (and others like it) to consider the blockchain-specific properties identified in Section 3.

TABLE 3 The smart contracts used for evaluating AGSOLT and their characteristics. Comm. stands for the "communication" domain.

Contract Name	Domain	# Statements	# Branches	Found	Sender Dep.	Value Dep.	Acc. as Vars	NE Acc. Dep.	Zero Acc. Dep.	Block Dep.	Time Dep.
AddressBook	Comm.	19	54	✗	✓	✗	✓	✗	✗	✗	✗
array-utils	Storage	144	257	✗	✗	✓	✗	✗	✗	✗	✗
BadAuction	Token	7	7	✗	✓	✓	✗	✗	✗	✗	✗
BasicToken	Token	11	8	✗	✓	✓	✓	✓	✗	✗	✗
Casino	Exploit	38	29	✗	✓	✓	✗	✗	✗	✗	✓
DateTime	Time	90	143	✗	✗	✗	✗	✗	✗	✗	✗
DosAuction	Exploit	7	7	✓	✓	✓	✗	✗	✗	✗	✗
EIP20Standard-Token	Token	24	13	✓	✓	✓	✓	✗	✗	✗	✗
EasyPayAnd-WithDraw	Token	7	8	✗	✓	✓	✗	✗	✗	✗	✗
EtherBank	Exploit	13	17	✗	✓	✓	✓	✗	✗	✗	✗
EzToken	Token	31	11	✓	✓	✓	✓	✗	✗	✗	✗
FixedSupplyToken	Token	39	22	✓	✓	✓	✓	✗	✗	✗	✗
FundRaising	Finance	23	21	✗	✓	✓	✗	✗	✗	✗	✓
Gift_1_ETH	Exploit	18	18	✓	✗	✓	✗	✗	✗	✗	✗
Greeter	Comm.	15	81	✗	✗	✗	✗	✗	✗	✗	✗
Greeter2	Comm.	13	60	✗	✗	✗	✗	✗	✗	✗	✗
Greeter3	Comm.	15	73	✗	✗	✗	✗	✗	✗	✗	✗
GuardCheck	Finance	10	14	✗	✓	✓	✓	✗	✓	✗	✗
GuessTheNum-berChallenge	Exploit	6	8	✗	✗	✓	✗	✗	✗	✗	✗
Identity	Identity	53	131	✗	✓	✓	✗	✗	✗	✗	✗
IdentityManager	Identity	49	90	✓	✓	✗	✓	✗	✗	✗	✗
LotteryFor10	Betting	45	44	✓	✓	✓	✗	✗	✗	✓	✗
LotteryMultiple-Winners	Betting	31	45	✗	✓	✓	✗	✗	✗	✗	✗
MultiSigWallet (1)	Wallet	56	70	✗	✓	✓	✓	✓	✗	✗	✗
MultiSigWallet (2)	Wallet	59	83	✗	✓	✓	✓	✓	✗	✗	✗
MyAdvancedToken	Token	53	3	✓	✓	✓	✓	✗	✗	✗	✗
OpenAddressLottery	Betting	30	34	✓	✓	✓	✓	✗	✗	✗	✗
PermissionGroups	Identity	58	86	✓	✓	✗	✓	✗	✓	✗	✗
Prover	Comm.	27	17	✓	✓	✗	✓	✗	✗	✗	✗
Randomness	Betting	22	17	✗	✓	✗	✗	✗	✗	✗	✗
Reentrance	Exploit	9	14	✓	✓	✓	✓	✗	✗	✗	✗
Rubixi	Exploit	56	102	✓	✓	✓	✓	✗	✗	✗	✗
SecureAuction	Finance	11	6	✓	✓	✓	✗	✗	✗	✗	✗
TestDateTime	Time	160	252	✓	✗	✗	✗	✗	✗	✗	✗
theRun	Exploit	62	83	✓	✓	✓	✓	✓	✗	✗	✗
VulnerableTwoStep	Exploit	11	10	✗	✓	✓	✗	✗	✗	✗	✗

5.2 | AGSOLT Evaluation

To evaluate the effectiveness of AGSOLT as well as compare the effectiveness of our random search and guided search, we perform an empirical study steered by the following research questions.

- **RQ1 (Effectiveness).** *Which is the coverage of the genetic algorithm approach compared to the random approach when generating test cases for Solidity smart contracts?*

- **RQ2 (Efficiency).** *Which is the execution time of the genetic algorithm approach compared to the random approach when generating test cases for Solidity smart contracts?*

- **RQ3 (Test Case Length).** *Which is the average number of statements in a test case for the genetic algorithm approach compared to the random approach when generating test cases for Solidity smart contracts?*

The first two research questions are selected because they give insight into the performance of the two approaches as well as the general performance of AGSOLT. The third research question is included because creating small, "*human-readable*" test cases is a secondary objective of DYNAMOSA [12].

To answer the research questions, we implement both the random search and DYNAMOSA-based guided search that were described in sections 4.2.1 and 4.2.2 and run AGSOLT for each approach and for each smart contract in Table 3 to generate a test suite until either *i*) full branch coverage is achieved or *ii*) the tool has gone back to the start of the search loop in Figure 1 100 times. We repeated the process ten times for each smart contract to account for the inherent randomness of both approaches. Our parameter settings for the genetic algorithm are the same as those used for evaluating DYNAMOSA [12], and the configurable options discussed in Section 4.3 were appropriately set whenever possible to constrain the search. In order to fairly compare the approaches and keeping with the above settings, we set the population size to 50 individuals for both approaches; therefore, the search budget consists of 5,000 test case evaluations or up to 200,000 method evaluations per smart contract. As previously mentioned, we used GANACHE to simulate the Ethereum blockchain, as it is much faster than a decentralized blockchain implementation. The execution was run on virtual machines running Ubuntu server with a RAM of 16GB. For each generated test case, we measure its branch coverage, the time spent running tests on the blockchain, the total time, and the number of statements. Additionally, we compute the statistical significance of the difference between the two approaches using *Wilcoxon's test* [48] with a p-value threshold of 0.05 as well as the Vargha-Delaney statistic ($\hat{A}_{12}$) [49] which is used to measure the magnitude of the difference.

5.3 | Analysis of the Results

First, all tables miss the results for three contracts, which returned an error. We found that invoking some functions of DATETIME and IDENTITY could cost more Gas than the block limit and that calling a function in IDENTITY with AGSOLT can produce an out of bounds error; therefore, we excluded them from the performance evaluation. However, we included these smart contracts in Table 3 since they demonstrate the usefulness of AGSOLT as a tool capable of detecting errors in popular real-world smart contracts.

5.3.1 | RQ 1. (Effectiveness)

Table 4 shows the mean branch coverage in terms of branches covered and the percentage of total branches covered for both DYNAMOSA [12] and the fuzzer approach. Overall, both approaches achieved good branch coverage, Table 5 shows that DYNAMOSA managed to achieve full branch coverage for 21, while the fuzzer achieves full branch coverage for 18 smart contracts. Full branch coverage could not be achieved for several reasons. For example, some branches may be infeasible, or AGSOLT settings should be tweaked further. For example, we noted that the "LotteryFor10" contract had one branch that was consistently not covered and found that this was because longer test cases (containing more than 40 statements) were necessary to cover this branch. One notable outlier on which both approaches perform poorly is the "theRun" contract, which relies on the block hash to simulate randomness, which is something that cannot be manipulated by AGSOLT.

Table 4 also reports *p*-values from a Wilcoxon test as well as the $\hat{A}_{12}$ and effect size from a Vargha-Delaney test comparing the distributions of the achieved branch coverages (in percentages) by applying the genetic and fuzzing approach each ten times per smart contract. Looking closer at the *p*-values and Varghay-Delaney statistic, we see that DYNAMOSA achieves significantly higher coverage ($p \leq 0.05$) than the fuzzer in six cases, each with large effect size. In contrast, the fuzzer significantly outperformed DYNAMOSA only once, also with large effect sizes. Additionally, when DYNAMOSA outperforms the fuzzer, the average branch coverage increases between 3% and 25%, while the fuzzer only achieves a 2% (2 branches) increase. We manually investigated those smart contracts for which the guided search could achieve full branch coverage, while the random search could not. In every case, we found that the branches not reached by the random search resulted from nested if-else statements and assertions. This observation is in line with existing literature [17, 31] that suggests that genetic algorithms could prove beneficial when compared to random testing approach for exercising deeper functionalities in code.

The genetic algorithm (i.e., DYNAMOSA) significantly outperformed the fuzzing algorithm when generating test cases for six Solidity smart contracts, whereas the opposite happened only once.

TABLE 4 Comparison for the achieved branch coverage for the genetic search algorithm and the fuzzing algorithm.

Name	# Branches	Mean Cov. Gen.		Mean Cov. Fuz.		p-val	$\hat{A}_{12}$	Effect Size
		#	%	#	%			
AddressBook	54	54.0	1.00	54.0	1.00	1.00	0.50	negligible
BadAuction	7	7.00	1.00	7.00	1.00	1.00	0.50	negligible
BasicToken	8	8.00	1.00	8.00	1.00	1.00	0.50	negligible
Casino	29	25.0	0.86	24.1	0.83	**0.03**	**0.75**	large
DosAuction	7	7.00	1.00	7.00	1.00	1.00	0.50	negligible
EIP20StandardToken	13	13.0	1.00	13.0	1.00	1.00	0.50	negligible
EasyPayAndWithDraw	8	8.0	1.00	6.00	0.75	**0.00**	**1.00**	large
EtherBank	17	14.0	0.82	14.0	0.82	1.00	0.50	negligible
EzToken	11	11.0	1.00	11.0	1.00	1.00	0.50	negligible
FixedSupplyToken	22	21.7	0.99	22.0	1.00	0.08	0.35	small
FundRaising	21	21.0	1.00	21.0	1.00	1.00	0.50	negligible
Gift_1_ETH	18	14.0	0.78	14.0	0.78	1.00	0.50	negligible
Greeter	81	81.0	1.00	81.0	1.00	1.00	0.50	negligible
Greeter2	60	60.0	1.00	60.0	1.00	1.00	0.50	negligible
Greeter3	73	73.0	1.00	73.0	1.00	1.00	0.50	negligible
GuardCheck	14	14.0	1.00	14.0	1.00	1.00	0.50	negligible
GuessTheNumberChallenge	8	8.00	1.00	8.00	1.00	1.00	0.50	negligible
IdentityManager	90	73.6	0.82	55.0	0.61	**0.00**	**1.00**	large
LotteryFor10	44	43.0	0.98	43.0	0.98	1.00	0.50	negligible
LotteryMultipleWinners	45	44.7	0.99	43.4	0.96	**0.05**	**0.78**	large
MultiSigWallet (1)	70	62.0	0.89	62.7	0.90	0.44	0.33	medium
MultiSigWallet (2)	83	76.2	0.92	74.5	0.90	0.33	0.69	medium
MyAdvancedToken	3	3.00	1.00	3.00	1.00	1.00	0.50	negligible
OpenAddressLottery	34	32.0	0.94	32.0	0.94	1.00	0.50	negligible
PermissionGroups	86	85.7	0.997	83.7	0.97	**0.01**	**0.96**	large
Prover	17	17.0	1.00	17.0	1.00	1.00	0.50	negligible
Randomness	17	16.0	0.94	16.0	0.94	1.00	0.50	negligible
Reentrance	14	13.0	0.93	13.0	0.93	1.00	0.50	negligible
Rubixi	102	67.0	0.66	69.0	0.68	**0.00**	**0.05**	large
SecureAuction	6	6.00	1.00	6.00	1.00	1.00	0.50	negligible
TestDateTime	252	243	0.96	240	0.95	**0.02**	**0.75**	large
theRun	83	34.0	0.41	34.0	0.41	1.00	0.50	negligible
VulnerableTwoStep	10	10.0	1.00	10.0	1.00	1.00	0.50	negligible

5.3.2 | RQ 2. (Efficiency)

Table 6 shows the average number of generations (including the (first) random initialisation) for both approaches as well as the mean total time spend and the average time per generation. Additionally the Chain Time column, shows the average percentage of time that was spend running the tests on our blockchain implementation (as opposed to evaluating- and generating new test cases). Interestingly, on average both approaches are more or less equally fast: with DynaMOSA taking 45.5 generations on average compared to 52.4 generations for the fuzzer and 5,683 seconds to 5,984 seconds for the fuzzer. This is surprising because the DynaMOSA algorithm follows the additional selection, crossover and mutation steps described in Section 4. One possible explanation for this is the preference criterion 2, which guides the search towards smaller test cases. Smaller test cases, in turn, take up less time; especially since Table 6 shows that most of the time in our experiments was used running the test cases on the blockchain. In order to properly compare the results for the two implementations we performed Wilcoxon tests and Vargha-Delany tests comparing the distributions of average run times for the smart contracts for each approach, the results of which are shown in the final 3 columns of Table 6.

TABLE 5 Frequency of full branch coverage for the two approaches.

Name	Full Cov. Gen.	Full Cov. Fuz.
AddressBook	10	10
BadAuction	10	10
BasicToken	10	10
DosAuction	10	10
EIP20StandardToken	10	10
EasyPayAndWithDraw	10	-
EzToken	10	10
FixedSupplyToken	7	10
FundRaising	10	10
Greeter	10	10
Greeter2	10	10
Greeter3	10	10
GuardCheck	10	10
GuessTheNumberChallenge	10	10
LotteryMultipleWinners	7	2
MultiSigWallet (1)	1	-
MyAdvancedToken	10	10
PermissionGroups	8	-
Prover	10	10
SecureAuction	10	10
VulnerableTwoStep	10	10

There are ten smart contracts for which DYNAMOSA significantly ($p \leq 0.05$) outperformed the fuzzer (9 with large and 1 medium effect size). The faster performance for "EasyPayAndWithDraw" and "PermissionGroups" can be attributed to the fact that for these smart contracts the genetic approach manages to regularly achieve branch coverage before the budget is consumed, whereas the fuzzer does not. For the other smart contracts we speculate that the preference criterion (2) in DYNAMOSA, which guides the search to smaller test cases, saves time when running the tests in the blockchain environment and evaluating their performance as described in sections 4.2.3 through 4.2.5. There are seven smart contracts for which the fuzzing approach significantly outperformed the genetic search (each with large effect size). For each of these, we see that the fuzzer, spends a smaller percentage off time **off**-chain compared to DYNAMOSA. This makes as the fuzzer bypasses the (computationally intensive) *selection*, *crossover* and *mutation* steps described in Section 4.2.2.

DYNAMOSA was significantly faster than the fuzzing algorithm on ten smart contracts, whereas the opposite happened seven times.

5.3.3 | RQ 3. Test Case Length

Table 7 shows the average test case length (in number of statements) for the final solution presented by both the genetic algorithm and the fuzzing algorithm. This solution is an archive, which stores for each branch to be covered, the shortest test case that covers it. Even though the creators of DYNAMOSA cite the use of a preference criterion as a means for reducing the size of the test cases in the final test suite, in this experiment implementing an archive resulted in fairly similar results, at first glance, compared to DYNAMOSA averaging 4.96 statements and the fuzzer averaging 5.03 statements.

To better compare the results of the two approaches a Wilcoxon test and a Vargha-Delany test were performed comparing the distributions of the average test case lengths of the final test suites for each smart contract. DYNAMOSA produced significantly shorter ($p \leq 0.05$ test cases for five smart contracts (four with large effect size and one with medium effect size), each of which it was also significantly faster for as shown in Table 6. This supports the theory that the smaller test cases found by the guided search can lead to an increase in efficiency when compared to a random search. The fuzzing approach yielded significantly smaller test cases in the final test suite for 4 smart contracts. For the "EasyPayAndWithDraw" smart contract this can be explained by

TABLE 6 Comparison for the time spend on creating tests for the genetic search algorithm and the fuzzing algorithm.

Name	Generations		Time/Generation		Total Time (s)		Chain Time (%)		p-value	$\hat{A}_{12}$	Effect Size
	Gen.	Fuz.	Gen.	Fuz.	Gen.	Fuz.	Gen.	Fuz.			
AddressBook	3.50	1.90	122	277	428	527	0.72	0.71	0.28	0.40	small
BadAuction	1.00	1.00	85.9	86.5	85.9	86.5	0.84	0.84	0.96	0.5	negligible
BasicToken	1.00	1.00	92.7	95.2	92.7	95.2	0.76	0.77	0.39	0.39	small
Casino	101	101	80.3	133	8115	13432	0.82	0.82	**0.01**	**0.00**	large
DosAuction	1.00	1.00	69.2	70.9	69.2	70.9	0.85	0.85	0.58	0.43	negligible
EIP20StandardToken	1.00	1.00	1001	110	101	110	0.76	0.75	**0.01**	**0.18**	**large**
EasyPayAndWithDraw	3.50	101	122	78.1	425	7888	0.84	0.86	**0.01**	**0.00**	large
EtherBank	101	101	67.8	19.5	6848	1970	0.84	0.88	**0.01**	**1.00**	large
EzToken	1.00	1.00	137	146	137	146	0.72	0.73	**0.05**	**0.21**	**large**
FixedSupplyToken	34.7	4.10	79.5	172	2758	704	0.79	0.77	0.33	0.60	small
FundRaising	1.00	1.00	81.1	79.7	81.1	79.7	0.77	0.77	0.80	0.52	negligible
Gift_1_ETH	101	101	66.4	93.1	67078	9399	0.83	0.82	**0.01**	**0.00**	large
Greeter	2.70	1.30	172	162	463	210	0.70	0.69	0.33	0.63	small
Greeter2	1.00	1.20	183	182	183	218	0.68	0.69	**0.03**	**0.28**	medium
Greeter3	2.10	1.60	172	250	361	400	0.71	0.68	0.28	0.20	large
GuardCheck	1.00	1.00	86.2	74.0	86.2	74.0	0.82	0.83	**0.02**	**0.75**	large
GuessTheNumberChallenge	1.90	1.30	52.3	35.5	99.5	46.2	0.84	0.75	0.33	0.23	large
IdentityManager	101	101	141	113	14196	11430	0.73	0.76	0.09	0.70	medium
LotteryFor10	101	101	149	113	15039	11389	0.73	0.79	**0.01**	**1.00**	large
LotteryMultipleWinners	65.9	91.7	174	95.1	11472	8721	0.76	0.79	0.28	0.72	medium
MultiSigWallet (1)	97.9	101	149	96.8	14600	9778	0.75	0.78	**0.01**	**1.00**	large
MultiSigWallet (2)	101	101	190	96.6	19210	9760	0.74	0.79	**0.01**	**1.00**	large
MyAdvancedToken	1.00	1.00	135	139	135	139	0.71	0.71	0.72	0.39	small
OpenAddressLottery	101	101	245	101	24729	10159	0.79	0.81	**0.01**	**1.00**	large
PermissionGroups	66.7	101	132	157	8827	15878	0.74	0.78	**0.01**	**0.10**	large
Prover	1.00	1.00	200	193	200	193	0.71	0.69	0.28	0.62	small
Randomness	101	101	86.3	86.5	8720	8740	0.82	0.83	0.80	0.51	negligible
Reentrance	101	101	59.6	93.0	6022	9391	0.84	0.83	**0.01**	**0.00**	large
Rubixi	101	101	53.4	121	5393	12257	0.70	0.71	**0.01**	**0.00**	large
SecureAuction	1.00	1.00	90.3	87.0	90.3	87.0	0.81	0.81	0.09	0.63	small
TestDateTime	101	101	211	345	21276	34882	0.62	0.62	**0.01**	**0.11**	large
theRun	101	101	104	91.8	10538	9269	0.75	0.80	**0.03**	**0.79**	large
VulnerableTwoStep	1.00	1.00	71.3	73	71.3	72.6	0.84	0.83	0.65	0.48	negligible
Mean	45.5	49.4	120	123	5683	5984	0.77	0.77	-	-	-

the guided search, which achieves full branch coverage fairly quickly, whereas the fuzzer consumes the full budget and thus has many more opportunities to generate smaller test cases. For the other smart contracts (two with large effect size and one with medium effect size) the improvement is very minor: ranging from 0.15 to 0.42 statements on average. If instead we look only at those smart contracts for which the fuzzing approach and the genetic approach complete in the same number of generations the average test case length for DynaMOSA becomes 3.70 statements and the average test case length for the Fuzzer becomes 3.96 which is slightly bigger.

The genetic algorithm (i.e., DynaMOSA) produced significantly smaller test cases in the final test suites when compared to the fuzzing algorithm for five smart contracts. The fuzzing algorithm produced significantly shorter test cases in the final test suites when compared to DynaMOSA for 3 smart contracts.

6 | THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our experiment.

TABLE 7 Comparison between the average test case length for the genetic search algorithm and the fuzzing algorithm.

Name	Gen.	Fuz	p-value	$\hat{A}_{12}$	Effect Size
AddressBook	9.39	10.1	0.39	0.44	negligible
BadAuction	2.94	3.09	0.34	0.45	negligible
BasicToken	3.67	3.54	0.61	0.50	negligible
Casino	5.88	9.1 0	**0.01**	**0.10**	**large**
DosAuction	3.66	3.79	0.84	0.46	negligible
EIP20StandardToken	5.71	5.89	0.58	0.42	small
EasyPayAndWithDraw	8.1	2.0	**0.01**	**1.00**	**large**
EtherBank	2.37	2.66	**0.04**	**0.12**	**large**
EzToken	4.47	4.66	0.61	0.41	small
FixedSupplyToken	4.15	4.85	0.28	0.41	small
FundRaising	6.55	6.61	0.88	0.48	negligible
Gift_1_ETH	2.02	2.01	0.32	0.60	small
Greeter	7.62	7.51	0.96	0.52	negligible
Greeter2	7.01	7.34	0.72	0.44	negligible
Greeter3	8.22	9.16	0.09	0.26	large
GuardCheck	4.46	4.71	0.44	0.40	small
GuessTheNumberChallenge	14.46	13.69	0.54	0.50	negligible
IdentityManager	3.44	3.24	0.44	0.44	negligible
LotteryFor10	4.49	4.46	0.72	0.48	negligible
LotteryMultipleWinners	8.33	7.15	0.07	0.71	medium
MultiSigWallet (1)	3.88	6.59	**0.01**	**0.10**	**large**
MultiSigWallet (2)	3.85	6.81	**0.01**	**0.07**	**large**
MyAdvancedToken	2.70	2.93	0.51	0.42	small
OpenAddressLottery	2.51	2.09	**0.02**	**0.80**	**large**
PermissionGroups	7.16	6.61	0.44	0.58	small
Prover	4.41	4.42	0.57	0.56	negligible
Randomness	2.46	2.58	0.07	0.28	medium
Reentrance	2.15	2.0	**0.05**	**0.70**	**medium**
Rubixi	3.73	3.52	0.58	0.50	negligible
SecureAuction	3.87	3.58	0.24	0.65	small
TestDateTime	2.42	2.14	**0.01**	**0.97**	**large**
theRun	2.1	2.17	**0.05**	**0.32**	**medium**
VulnerableTwoStep	5.33	5.15	0.80	0.52	negligible
Mean	4.96	5.03			

Construct Validity. We demonstrated that transaction properties, blockchain properties, and interactive properties are present in some of the most popular Solidity smart contracts on Github. Additionally, we showed the effectiveness and efficiency of AGSOLT by comparing a search-based test approach with a random testing one in terms of branch coverage, execution time, and test case length. Both approaches were implemented in the same tool (i.e., AGSOLT) and executed on the same hardware environment to make the comparison as fair as possible. We acknowledge that implementation issues could negatively impact the final results. However, please consider that we strictly followed the definition of the algorithm provided by Panichella *et al.* [12] and that our implementation is publicly available to allow other researchers to replicate our study.

Internal Validity. All the experiments were executed ten times to address the inherent randomness of both approaches. Fine-tuning the parameters of the DynaMOSA algorithm [12] could also have affected the internal validity of the experiments; since setting these parameters is challenging [47], we used the default values suggested by the creators of the algorithm [12].

External Validity. We tested AGSOLT on a set of real-world smart contracts from a wide variety of developers. We also ensured that each basic variable type and arrays in Solidity were included in the data set. Despite exhibiting each of the properties that are indicative of the identified blockchain-specific challenges it is still possible that our data set is not representative of Solidity smart contracts in general and in general our results depend on the assumption that this data set is representative. Future experimentation with a larger data set is desirable. AGSOLT cannot yet handle user-defined input variable types nor smart contracts that rely on previously deployed smart contracts for their initialization. Adding this feature is part of our research agenda. Our conclusions are derived from the results obtained only on one genetic algorithm, namely DynaMOSA [12]. Our research agenda includes experimentation with a broader set of search algorithms. We did not run the test cases in a distributed blockchain, but we relied on GANACHE, a framework to run tests, execute commands, and inspect smart contracts. However, please consider that the resulting test suites are presented conveniently and can be easily used in any test network (e.g., Ropstein[13]).

Conclusion Validity. The results were obtained by repeating the experiments enough times and adopting appropriate statistical tests to draw valid conclusions. Specifically, we used the Wilcoxon test [48] to test the significance of the differences and the Vargha-Delaney statistic [49] to estimate the effect size of the observed differences.

7 | CONCLUSION

This paper discussed the challenges that arise when applying automated test case generation in a blockchain environment, identifying three different categories: transaction properties, blockchain properties and interactive properties. We presented, explored and partially validated AGSOLT, a tool that addresses these challenges and creates test suites that aim to achieve branch coverage for Solidity smart contract unit testing.

AGSOLT works with both a random testing approach (i.e., a fuzzer) and a guided-search approach (i.e., the DYNAMOSA genetic algorithm [12]). We gathered a data set consisting of real-world smart contracts from GitHub. We demonstrated that many of these contracts exhibit behaviors that align with the challenges we identified. Additionally, we have shown the effectiveness and efficiency of AGSOLT by achieving good branch coverage with both approaches. In doing so we presented the first comparison between a guided search and a random search in the domain of automated test case generation for smart contracts. We found that the DYNAMOSA algorithm outperformed our fuzzer for achieving branch coverage, but ascertained that neither approach is significantly faster or produces significantly smaller test cases for the final test suite. The fact that the fuzzer was not faster, despite not going through the extra steps of selection, cross-over and mutation, is interesting and deserves further investigation. We hypothesize that this could be due to the preference criterion of DYNAMOSA, which should, in theory, result in less time spent on the execution and evaluation steps of the testing procedure. Finally, remarkably, we have shown that three of the most prevalent smart contracts on Github, suffer from critical failures (crashes) that emerged during our tests, demonstrating the potential real-world value of AGSOLT.

In our future agenda, we plan to extend our current baseline a larger set of commercial smart contracts. Moreover we intend to leverage the parameterization of our testing approach with more search algorithms, e.g., the neural machine transaltion-based approach of Tufano et. al. [50]. Additionally, we will expand AGSOLT to test inter-contract dependencies with the final goal of creating test cases in blockchain environments when multiple smart contracts interact.

References

1. Anderson L, Holz R, Ponomarev A, Rimba P, Weber I. New kids on the block: an analysis of modern blockchains. *arXiv preprint arXiv:1606.06530* 2016.

2. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: ; 2016: 254–269.

3. Delmolino K, Arnett M, Kosba A, Miller A, Shi E. Lab: Step by Step towards Programming a Safe Smart Contract. 2015.

4. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts.. *IACR Cryptology ePrint archive* 2016; 2016: 1007.

[13] https://ethereum.org/en/developers/docs/networks/#testnet-faucets

5. Jiang B, Liu Y, Chan W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: ; 2018: 259–269.

6. Daka E, Campos J, Fraser G, Dorn J, Weimer W. Modeling readability to improve unit tests. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* 2015: 107–118. doi: 10.1145/2786805.2786838

7. Grano G, Scalabrino S, Gall HC, Oliveto R. An empirical investigation on the readability of manual and generated test cases. *Proceedings - International Conference on Software Engineering* 2018(May): 348–351. doi: 10.1145/3196321.3196363

8. Almasi MM, Hemmati H, Fraser G, Arcuri A, Benefelds J. An industrial evaluation of unit test generation: Finding real faults in a financial application. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017* 2017: 263–272. doi: 10.1109/ICSE-SEIP.2017.27

9. Zou W, Lo D, Kochhar PS, et al. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 2019; 5589(March 2018): 1–1. doi: 10.1109/tse.2019.2942301

10. Wang X, Wu H, Sun W, Zhao Y. Towards Generating Cost-Effective Test-Suite for Ethereum Smart Contract. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering* 2019: 549–553. doi: 10.1109/SANER.2019.8668020

11. Harman M, Jia Y, Zhang Y. Achievements, Open Problems and Challenges for Search Based Software Testing. In: ; 2015: 1-12.

12. Panichella A, Kifetew FM, Tonella P. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 2017; 44(2): 122–158.

13. Fraser G, Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering* 2012; 39(2): 276–291.

14. Shamshiri S. Automated Unit Test Generation for Evolving Software. *ESEC/FSE 2015* 2015: 1038–1041.

15. Yoo S, Harman M. Regression testing minimization, selection and prioritization: asurvey. *Software Testing Verification and Reliability* 2010(January 2010): 1–22. doi: 10.1002/stvr

16. Arcuri A, Iqbal MZ, Briand L. Black-box system testing of real-time embedded systems using random and search-based testing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2010; 6435 LNCS: 95–110. doi: 10.1007/978-3-642-16573-3_8

17. Shamshiri S, Rojas JM, Gazzola L, et al. Random or evolutionary search for object-oriented test suite generation?. *Software Testing Verification and Reliability* 2018; 28(4): 1367–1374. doi: 10.1002/stvr.1660

18. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system.. 2008.

19. Buterin V. Ethereum: a next generation smart contract and decentralized application platform.. 2013.

20. Wood G, others . Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 2014; 151(2014): 1–32.

21. Castillo dM. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft.. .

22. Zou W, Lo D, Kochhar PS, et al. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 2019: 1-1.

23. Anand S, Burke EK, Chen TY, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 2013; 86(8): 1978–2001.

24. Zhang P, Xiao F, Luo X. SolidityCheck: Quickly Detecting Smart Contract Problems Through Regular Expressions. *arXiv preprint arXiv:1911.09425* 2019.

25. Wu H, Wang X, Xu J, Zou W, Zhang L, Chen Z. Mutation testing for ethereum smart contract. *arXiv preprint arXiv:1908.03707* 2019.

26. Zhang P, Yu J, Ji S. ADF-GA: Data Flow Criterion Based Test Case Generation for Ethereum Smart Contracts. *arXiv preprint arXiv:2003.00257* 2020.

27. Liu Y, Li Y, Lin SW, Yan Q. ModCon: A Model-Based Testing Platform for Smart Contracts. In: ; 2020.

28. Sutton M, Greene A, Amini P. *Fuzzing: brute force vulnerability discovery*. Pearson Education . 2007.

29. Smith J. Echidna, a smart fuzzer for Ethereum. 2018.

30. Bartoletti M, Pompianu L. An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Springer. ; 2017: 494–509.

31. Harman M, McMinn P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *Software Engineering, IEEE Transactions on* 2010; 36: 226 - 247. doi: 10.1109/TSE.2009.71

32. Mitchell M. *An introduction to genetic algorithms*. MIT press . 1998.

33. Laumanns M, Thiele L, Deb K, Zitzler E. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary computation* 2002; 10(3): 263–282.

34. Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 2002; 6(2): 182–197.

35. Scalabrino S, Grano G, Di Nucci D, Oliveto R, Lucia dA. Search-based testing of procedural programs: Iterative single-target or multi-target approach?. *Lecture Notes in Computer Science* 2016; 9962 LNCS: 64–79. doi: 10.1007/978-3-319-47106-8_5

36. Rojas JM, Vivanti M, Arcuri A, Fraser G. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 2017; 22(2): 852–893.

37. Fraser G, Arcuri A. The seed is strong: Seeding strategies in search-based software testing. In: IEEE. ; 2012: 121–130.

38. Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1987; 9(3): 319–349.

39. Lengauer T, Tarjan RE. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1979; 1(1): 121–141.

40. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 2011; 38(2): 278–292.

41. Tonella P. Evolutionary testing of classes. In: . 29. ACM. ; 2004: 119–128.

42. Von Lücken C, Barán B, Brizuela C. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational optimization and applications* 2014; 58(3): 707–756.

43. Panichella A, Kifetew FM, Tonella P. LIPS vs MOSA: A replicated empirical study on automated test case generation. In: Springer. ; 2017: 83–98.

44. Korel B. Automated software test data generation. *IEEE Transactions on software engineering* 1990; 16(8): 870–879.

45. Baresi L, Miraz M. Testful: Automatic unit-test generation for java classes. In: . 2. IEEE. ; 2010: 281–284.

46. Köppen M, Yoshida K. Substitute distance assignments in NSGA-II for handling many-objective optimization problems. In: 2007 (pp. 727–741).

47. Arcuri A, Fraser G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 2013; 18(3): 594–623.

48. Conover WJ, Conover WJ. *Practical nonparametric statistics*. Wiley New York . 1980.

49. Vargha A, Delaney HD. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 2000; 25(2): 101–132.

50. Tufano M, Drain D, Svyatkovskiy A, Deng SK, Sundaresan N. Unit test case generation with transformers. *arXiv* 2020.