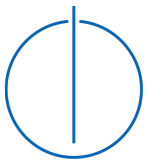
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
DEPARTMENT OF COMPUTER SCIENCE

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Fuzzing Algorand Smart Contracts

Denis Paluca



SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
DEPARTMENT OF COMPUTER SCIENCE

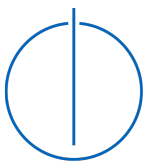
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Fuzzing Algorand Smart Contracts

Fuzzing von Algorand Smart Contracts

Author:	Denis Paluca
Supervisor:	Prof. Alexander Pretschner
Advisor:	Stephan Lipp
Submission Date:	15.10.2023



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.10.2023 Denis Paluca

Acknowledgments

First and foremost I would like to thank my advisor Mr. Stephan Lipp for their invaluable guidance and support throughout the course of this thesis. Your advice has been the corner stone of this work.

I would also like to thank Prof. Alexander Pretschner, the TUM Algorand Center of Excellence, and the Algorand Foundation for providing me with the opportunity to work on this thesis.

On a personal note, I would like to thank my family, my parents Ndue and Nikoleta, and my brother Damian. Thank you for your endless support and encouragement throughout my academic journey.

Abstract

Smart contracts, which are immutable programs that automate agreements between parties, are critical components in many blockchain networks. Their increasing role in sensitive financial transactions has heightened the importance of their security. However, traditional software testing methodologies, such as unit testing, have proven inadequate in improving smart contract security. These methods often test the functionality in isolation and do not account for the complex interactions between smart contracts and the blockchain. Fuzzing, a software testing technique where random inputs are injected into a program to uncover bugs, has emerged as a promising solution for enhancing smart contract security. While fuzzing techniques have faced challenges in adapting to smart contracts' distinct nature, such as the difficulty of defining bug oracles, property-based fuzzing tools such as Echidna have shown promise for Ethereum smart contracts. With the emergence of newer blockchain platforms like Algorand, which offers advantages like high throughput, low latency, and lower transaction fees, there is a limited number of specific tooling to ensure secure smart contracts. This thesis introduces AlgoFuzz, a property-based fuzzing tool tailored for Algorand smart contracts. The tool uses greybox fuzzing techniques to maximize code coverage and the discovered state space (the set of states reached during fuzzing). To evaluate the efficacy of AlgoFuzz, it is examined through experiments on Algorand smart contracts translated from the Solidity contracts in Echidna benchmarks. The experiment ran 10 times on 10 minutes intervals for 6 different configurations of the fuzzer on 12 contracts. Aside from that, we evaluate AlgoFuzz on two larger Algorand smart contracts, one of which is translated from the USD Tether smart contract and the other is a smart contract for a token exchange. In this experiment differently from the previous one, each run was 30 minutes. For the Echidna benchmarks, AlgoFuzz was able to cover 64.04% of the code, discover 2.66 unique coverage paths, and make 0.4 unique state transitions per call on average. For the two larger contracts, these values were 72.56%, 11.14, 0.2 respectively.

Contents

1. Introduction

In recent years, blockchain technology has gained immense traction, revolutionizing various industries because of its decentralized and transparent nature [1]. The rapid growth of blockchain technology has brought a new era of Decentralized Applications (dApps), which are made possible by smart contracts. Smart contracts are one of the key components of most modern blockchain networks. They are self-executing programs that run on the blockchain and are used to automate the execution of agreements between two or more parties [2]. As smart contracts handle an increasing set of sensitive transactions, ranging from financial agreements to supply chain management, making them secure has become a top priority. The financial value locked in these smart contracts coupled with the fact that they are immutable (meaning their code cannot be changed once deployed), makes exploiting their vulnerabilities a lucrative target for attackers. In the recent past, we have witnessed several high-profile attacks on smart contracts, which have resulted in the loss of billions of dollars in value [3]. One such attack was the Wormhole hack, which resulted in the loss of $320 million worth of cryptocurrency [4].

As with regular software, different software testing techniques have been proposed and are being actively used to detect vulnerabilities in smart contracts. Traditional software testing methodologies such as unit testing, have shown to be minimally effective in improving the security of smart contracts when used on their own [5]. Unit tests are only as good as the test cases written by the developer, and it is impossible to write test cases for a wide range of input scenarios. For this reason, fuzzing has been proposed as a complementary technique to unit testing.

Fuzz testing or fuzzing is a software testing technique that involves providing invalid, unexpected, random or semi-random data as inputs to a program [6]. The goal of fuzzing is to trigger unexpected behavior in the program, which may indicate the presence of a vulnerability. During fuzzing the program is monitored for problems such as crashes, assertion failures, and memory leaks. Most fuzzers require little to no knowledge of the program under test, making them easy to use. Also, considering the automated nature of fuzzing, it is simple to integrate it into the software development

lifecycle. It can be set up to continuously test new code changes without manual intervention, making it a good addition to the Continuous Integration (CI) and Continuous Delivery (CD) pipelines [7]. Fuzzing has achieved great success in detecting security flaws, and because of this, it has become the most widely used technique to discover vulnerabilities [8, 9].

Fuzzing has been used to detect vulnerabilities for smart contracts as well. However, adapting fuzzing techniques to smart contracts has presented its own set of challenges. Smart contracts are distinct from traditional programs because concerns such as crashes, assertion failures, and memory leaks typically do not impact them. If such issues arise, the transaction is simply reverted, ensuring the blockchain's state remains unchanged. For this reason, most existing fuzzers for smart contracts focus on detecting some specific common vulnerabilities, such as *reentrancy* where a function call is exploited to re-run multiple times before finishing. A different approach is to allow the user to specify a set of properties that the fuzzer should check for. This approach is called property-based fuzzing. It requires the user to have a good understanding of the smart contract and the vulnerabilities that it may contain. Different property-based fuzzing tools such as Echidna, have shown to be effective in detecting real-world vulnerabilities [10, 11].

Research Gap

Similar to Echidna, most of the existing work on smart contract fuzzing has focused on the Ethereum blockchain since it is the most widely used blockchain for smart contracts [12]. Novel blockchain platforms that support smart contracts and address the shortcomings of Ethereum have emerged in recent years. One such platform is the Algorand blockchain [13]. It uses a unique consensus mechanism called Pure Proof of Stake (PPoS), where users are randomly and secretly selected to propose new blocks. The likelihood that a user will be selected and the weight of its proposals and votes, are directly proportional to its stake. With this consensus mechanism, Algorand has been able to achieve high throughput and low latency while maintaining decentralization. Different from Ethereum, Algorand provides immediate transaction finality, which means that once a block is added to the blockchain, it cannot be reverted and that no forks can occur. Transaction fees in Algorand are also significantly lower than in Ethereum. These features have made Algorand an attractive platform for smart contracts. However, the lack of a mature tooling ecosystem for Algorand smart contracts has made it difficult for developers to write secure smart contracts. Static analyzers, such as Tealer [14] and Panda [15], also testing frameworks, such as AlgoBuilder [16],

have been introduced but there is still a lack of fuzzing tools for Algorand smart contracts.

Aims and Objectives

In this thesis, we propose a novel fuzzing tool designed specifically for Algorand smart contracts called *AlgoFuzz*. AlgoFuzz is a smart contract fuzzer with features such as structured input generation, property-based testing, assertion testing, and multiple fuzzing strategies. Our tool tests Algorand smart contracts in a dockerized environment running a local sandbox version of an Algorand node. AlgoFuzz leverages *greybox* fuzzing techniques such as generating inputs that maximize not only code coverage but also the discovered state space of the smart contract. Code coverage, the percentage of code that is executed, has been show to correlate with the number of bugs found [17]. Discovered state space, the unique set of states reached during execution, is also important for a property-based fuzzer because the more states it reaches, the higher the chance of finding an invalid state.

The main goal of this project is the development of AlgoFuzz, which will serve as a prototype in the field of Algorand smart contract fuzzing. We also evaluate the effectiveness of AlgoFuzz through a series of empirical experiments. The experiments are conducted on a set of Algorand smart contracts which are translated from the Ethereum smart contracts used in the Echidna benchmark suite. AlgoFuzz is also evaluated on two larger Algorand smart contracts, one of which is translated from the USD Tether [18] smart contract and the other is a smart contract for a token exchange.

Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2, we provide the necessary background information on blockchain, smart contracts, smart contract security, Algorand, fuzzing and previous related work. Moving to Chapter 3, we outline the methodology behind AlgoFuzz's development, discussing our design choices, the features and constraints that influenced these decisions, and conclude with a relevant case study. Chapter 4 lays out the research questions, sets up the experimental environment, and presents the results of the experiments. These results are then discussed in Chapter 5 in the context of the research questions. Furthermore, in the same chapter, we discuss the limitations of our work by providing possible threats to validity. Finally, in Chapter 6, we conclude the thesis by summarizing our findings and discussing possible future work.

2. Background and Related Work

This chapter introduces the technical background and related work for the thesis. It establishes the foundation for the thesis by introducing the concepts of blockchain, smart contracts and specifically smart contracts on the Algorand blockchain network. Furthermore, challenges related to smart contract security and testing are explored, underlining the importance of fuzzing as a relatively novel testing tool in the space. Existing projects in the field of smart contract fuzzing are also examined.

2.1. Blockchain

Bitcoin was first introduced in 2008 by an anonymous person or group of people under the name of Satoshi Nakamoto [19]. The purpose of the project was to facilitate online payments without having to go through a financial institution. Previous solutions were hindered by the double-spend problem, where a user can counterfeit the currency if there is no trusted central authority keeping track of the different transactions and the balances. Bitcoin solved this problem with a peer-to-peer network. In this network the nodes keep track of all the transactions that have been committed in a public ledger. First the transactions are broadcasted to the network and then they are collected in a pool called the *mempool*. These transactions are then added into blocks which are chained together through cryptographic hashes. This way, the ledger is immutable meaning that once a transaction is added to the ledger it cannot be removed or modified. The nodes in the network keep track of the longest chain of blocks and accept the longest chain as the valid one. This is called the Proof of Work (PoW) consensus mechanism. These nodes are incentivized to keep the network secure by receiving a reward in the form of newly minted bitcoins for every block they add to the chain. This process is called *mining* and the participants of this process are called *miners*. The PoW consensus is computationally expensive, which makes it difficult for an attacker to create a longer chain than the honest nodes. The attacker would have to have more computational power than the rest of the network combined. This is known as the 51% attack. The Bitcoin network has been running for over 10 years and has never been

compromised. There are several limitations to this network, such as low transaction throughput and high energy consumption of the PoW consensus mechanism. These limitations have led to the development of new blockchain networks with different consensus mechanisms.

Another interesting feature of Bitcoin that does not have much notoriety is Bitcoin Script. Bitcoin Script is a simple stack-based programming language that is used to define the conditions under which a transaction can be spent [20]. The execution of the script is successful if the stack is empty and failed otherwise. Bitcoin Script is not Turing-complete, which means that it cannot be used to implement arbitrary programs. It was not intended to be used for more complex scripts, but it did inspire the creation of new networks that would allow for more complex scripts, later called smart contracts.

2.2. Smart Contracts

The term "Smart Contract" was first coined by Szabo in 1996, to refer to promises between parties and the protocols to perform on these promises stored in digital form [21]. This concept was then first connected with blockchain through Ethereum which was proposed by Buterin in 2014 [22]. Similar to Bitcoin, at the time of its conception Ethereum was also running on a PoW consensus, and it had its own currency named Ether. Ethereum went on to expand on the concept of Bitcoin Scripts by allowing arbitrary Turing-complete code to be run on its network. This allowed for the creation of more complex smart contracts, similar to what Szabo had described earlier. Since the Ethereum network is completely permissionless (meaning anyone can participate without explicit permission from an authority), the addition of smart contracts transformed this network into a "world computer".

The model of the Ethereum smart contracts is simple and since Ethereum was the first blockchain network that supported smart contracts many succeeding networks are more or less similar. For this reason, it is interesting to mention how Ethereum smart contracts work and later to mention how Algorand Smart Contracts differ. Each smart contract has its state (the data of the contract), functions (the code of the contract) and an Ethereum account connected to this contract. The state of the contract is stored in the Ethereum blockchain and is accessible for anyone to read. The account of the contract is not controlled by a user but only by the code of the corresponding contract. The functions of the contract are the code that is executed when the contract is called. These functions have access not only to the state of the contract, but also to the state

of the blockchain. This means that the functions can read the state of other contracts and even call their functions. When a function is marked with `payable` it indicates that the function can receive ether. Also, the function has access to the address of the caller and the amount of ether that the caller may have sent. When a function is called, it is executed by all the nodes in the network. This means that the function is executed in a deterministic way and the result is the same for all the nodes. Because of this, the execution of the function can be expensive for the nodes. Therefore, the caller of the function has to pay a fee for the execution. This fee is calculated based on the amount of computational resources that the function consumes. This means that the more complex the function is, the more expensive it is to execute. The computational effort is measured in *gas* and the fees, called *gas fees*, are calculated by multiplying the gas with the gas price. The fee is paid in ether and is collected by the nodes that execute the function as an incentive. The fee is also used to prevent denial-of-service attacks, where a malicious user would try to execute a function that consumes a lot of resources and thus make the network unusable.

Allowing anyone to deploy and run arbitrary code on the network has led to the creation of a large ecosystem of dApps [23]. These dApps are similar to traditional web applications, but they are not controlled by a single entity. Instead, they are controlled by the code that is deployed on the network. Also, once deployed the code of these smart contracts cannot be changed due to the immutability property of the blockchain. This is a double-edged sword, as it means that no code fixes can be applied if there is a bug or a security vulnerability. Usually smart contracts control large sums of digital currencies or digital assets, which makes them a very attractive target for attackers. For these two reasons combined, it is of utmost importance to ensure the security of the smart contract before they are deployed.

2.3. Smart Contract Security

As we already laid out the importance of smart contract security, we will now explore the different types of vulnerabilities that can be found in smart contracts. Keeping smart contracts secure is compromised of two parts: writing vulnerability free smart contract code and also ensuring that off-chain operations (such as managing an administrating account) are conducted safely. We will be focusing on the first part.

Smart contracts have a lot of idiosyncrasies that set them apart from usual computer programs. Therefore, the types of vulnerabilities that we face when developing smart contracts are usually specific to the world of blockchain and they often differ from one

blockchain network to the other. The following is a non-exhaustive list of the most common vulnerabilities that can be found in smart contracts [24]:

- **Reentrancy** - One of the most iconic vulnerabilities, which has lead to the loss of millions of dollars in cryptocurrencies [25]. Functions in Ethereum are not executed atomically. This means that the execution of a function can be interrupted by another function call. This can lead to a vulnerability where a malicious user can call the same function multiple times before the first execution is finished. In this manner, the malicious user could be able to drain the funds of the contract.
- **Integer Overflow and Underflow** - This vulnerability is not specific to smart contracts, but it is prevalent in smart contracts due to the fact that most networks use fixed-size integers. Usually the maximum value of these integers is 2^{256}-1. If the result of an operation on integers is larger than the maximum value, the result will overflow and the integer will wrap around to 0. A malicious actor could find a way to exploit this vulnerability.
- **Timestamp Dependence** - The execution environment of the smart contract is managed by the miner of the block. If some logic of the contract depends on the timestamp variable, the miner can manipulate this variable to his advantage.
- **Front-running** - Every transaction on blockchain networks first becomes visible in the mempool prior to being executed. This visibility enables those on the network to view and act on a transaction before it is added to a block. Such exposure can be exploited by malicious actors to manipulate how a transaction is carried out.
- **Resource Limit Vulnerabilities** - The execution of a function in most blockchains will incur a cost to the caller. If a function is too complex, the costs of executing it might exceed certain limitations of the network or the funds of the contract. This can lead to certain functions of the contract being unusable after a while. An example of this would be calling a function that pays out funds to users from a large array. If the users in the array are too many, the function might not be able to pay out all the users due to the cost of the function exceeding the funds of the contract.
- **Simple Logical Errors** - Although most smart contract are thoroughly scrutinized by the community, still a significant number of vulnerabilities are caused by simple logical errors. Meaning errors that are specific to the logic of the contract and not common to all smart contracts. These errors cannot be avoided by adding features to your blockchain, as they are inherent in all computer programs written by humans.

To deal with these vulnerabilities different tools have been proposed such as static analysis tools, symbolic execution tools and fuzzers, which we will discuss more of on section 2.5.

In the next section we will explore the Algorand blockchain network, its smart contracts, how it handles the previously mentioned vulnerabilities and which vulnerabilities are relevant to it.

2.4. Algorand

Algorand was initially proposed in 2017 by Chen et al. [26]. It sought to improve on previous networks by solving the scalability trilemma, which states that a blockchain network can only have two of the following three properties: security, scalability and decentralization. This is due to the fact that previous networks relied on a consensus mechanism that was computationally expensive and thus limited the transaction throughput of the network. Algorand solved this problem by introducing a new consensus mechanism called PPoS. The PPoS consensus algorithm randomly selects a small set of validators for each block from a large pool of participants. These validators propose and vote on the next block, with the weight of their vote proportional to the amount of cryptocurrency they have staked. To ensure unpredictability and fairness, a cryptographic lottery determines the validators. This consensus mechanism has the advantage of being computationally inexpensive and thus allows the network to scale. The network is permissionless, which means that anyone can participate in the consensus. The consensus mechanism is also Byzantine fault tolerant, meaning the network can still function even if up to 1/3 of the nodes in the network are malicious. According to the authors, the possibility of a fork in the network is 10^{-18}, meaning the network practically achieves instant finality of a transaction, from the moment it enters the ledger. This is different from Bitcoin where the finality of a transaction is probabilistic and it increases as more blocks are added to the chain. For Bitcoin finality is usually considered reached after 6 confirmations (meaning 6 blocks after the one containing the interesting transaction). At the time of this writing, new blocks are produced at an average of 3.5 seconds and can hold up to 25000 transactions, which results in a transaction throughput of 7150 Transactions per Second (TPS) [27].

The native currency of the Algorand network is called Algo, and it has a maximum supply of 10 Billion Algos. The smallest unit of Algo is called a microAlgo and it is equal to 0.000001 Algo. Algo is used to pay the transaction fees of the network and also to participate in the consensus. Different from the Ethereum gas fee, Algorand fees

are only paid if the transaction is included in a block. Transaction fees are calculated based on congestion of the network, but because of the high TPS most transactions can use the minimum base fee of 0.001 Algos. In addition to the fees, Algorand has also Minimum Balance Requirement (MBR) for its accounts. The MBR acts as a deposit to rent space on the blockchain. Every time the amount of stored data on the blockchain increases, the MBR of the associated account also increases. When the space is liberated by deleting data, the MBR is also decreased. All accounts have a base MBR of 0.1 Algo.

One of the most interesting features of the Algorand network is the ability to perform an *Atomic Transfer* [28]. This way, up to 16 transactions are grouped in a transfer where they all succeed or fail together. This is useful in cases where a user wants to perform multiple transactions that are dependent on each other. Atomic transfers can also be used to allow complete strangers to trade. All the required transactions can be added into an atomic transfer and then be independently signed by all the participants. When all the participants have signed the transfer, it can be submitted to the network and all the transactions will be executed. This feature is also relevant to the way Algorand handles payments to smart contracts.

A powerful feature of the protocol is the ability to rekey an account. This means being able to change the private key, which signs transactions for an account but keep the same public key. This is useful in cases where the user wants to maintain a static public address but wants his private key rotated for security reasons. Rekeying an account is simply accomplished by issuing a transaction with the `rekey-to` field set to the public address of the new authoritative account.

Different from other networks, the ability to create fungible and non-fungible tokens is directly built into the protocol [29]. These tokens are created using Algorand Standard Asset (ASA) and a special transaction type is required when creating them. On other blockchains a smart contract is required to represent these tokens. Similarly, in Algorand you could have a smart contract to create and control the tokens, but it is not mandatory.

Smart signatures are another feature of the Algorand network, which should not be confused with smart contracts. They are similar to the Bitcoin Scripts. They contain logic that is used to sign a transaction. This logic is then submitted together with the transaction. The nodes in the network execute the smart signature and if the execution is successful, the transaction is signed and submitted to the network.

Algorand Smart Contracts

On the Algorand network smart contracts are written with the Transaction Execution Approval Language (TEAL), which is a Turing-complete assembly-like language that is interpreted by the Algorand Virtual Machine (AVM) [30]. Since TEAL is a low-level language, it is difficult for developers to write smart contracts directly in it. For this reason, different high-level solutions that compile to TEAL have been developed, such as Reach [31], PyTeal [32], Beaker [33] and TEALScript [34].

```python
from pyteal import *

handle_creation = Seq(
    App.globalPut(
        Bytes("Count"),
        Int(0)
    ),
    Approve()
)

router = Router(
    "example-contract",
    BareCallActions(
    no_op=OnCompleteAction
        .create_only(
            handle_creation
        ),
    ),
)

@router.method
def increment() -> Expr:
    return Seq(
        App.globalPut(
            Bytes("Count"),
            App.globalGet(
                Bytes("Count")
            ) + Int(1)
        )
    )
```

(a) PyTeal

```
#pragma version 8
txn NumAppArgs
int 0
==
bnz main_l4
txna ApplicationArgs 0
method "increment()void"
==
bnz main_l3
err
refs: 1
main_l3:
    txn OnCompletion
    int NoOp
    ==
    txn ApplicationID
    int 0
    !=
    &&
    assert
    callsub increment
    int 1
    return
refs: 1
main_l4:
    txn OnCompletion
    int NoOp
    ==
    bnz main_l6
    err
refs: 1
main_l6:
    txn ApplicationID
    int 0
    ==
    assert
    byte "Count"
    int 0
    app_global_put
    int 1
    return
refs: 1
increment:
    proto 0 0
    byte "Count"
    byte "Count"
    app_global_get
    int 1
    +
    app_global_put
    retsub
```

(b) TEAL

Figure 2.1.: Example PyTeal code compiled to TEAL.

The TEAL contracts can be deployed and remotely called from any node in the Algorand

blockchain. An on-chain instantiation of an Algorand smart contract is called an *Application*. Special transactions, named *Application Call transactions*, are required to trigger the different functions of the application. Similar to Ethereum smart contracts, an application can modify state, access on-chain values and execute transactions. Each application has an associated account, that only the application can control.

An application can store state in three different ways [35]:

- **Global Storage** - This is stored on the application itself and only the application can modify it. It allows for the storage of up to 64 key-value pairs. The amount of required storage must be determined on creation and it cannot be changed afterwards.
- **Local Storage** - Similarly to global storage, it is also determined on creation and cannot be change. The maximum amount of key-value pairs is 16. The difference is that local storage is stored on the account that called the application. The local storage of an account is allocated after an account submits an *opt-in* application call. The user can also remove the local storage from his account by submitting a *clear state* application call.
- **Box Storage** - Box storage is similar to global storage, the only difference being box storage can be dynamically allocated. The box id is required when making an application call for the box storage to be allocated or to be read.

Since the state is stored on the blockchain, it is publicly available to anyone. This means it can be read by clients but also by other applications. This allows for the creation of more complex applications that interact with each other.

Smart contracts are implemented with two independent programs: the *Approval Program* and the *Clear State Program*. When making an application call transaction, one can choose form the following transaction types [36]:

- **NoOp** - Generic application call to execute the Approval Program.
- **OptIn** - Begins participation and enables local storage for an application.
- **DeleteApplication** - Deletes the application and all associated data.
- **UpdateApplication** - Updates the TEAL programs for the application. Global and local state requirements cannot be updated and the current state does not change.
- **CloseOut** - Account can end their participation on the contract and clear the local storage. This call can fail and the account can remain opted-in.

- **ClearState** - Similar to CloseOut, but the account will be opted-out regardless if the call succeeds or fails. In other words, the local state of the account associated with this application will be forcefully cleared.

Contracts often require data stored within the ledger. To maintain a high level of performance, the AVM restricts how much of the ledger can be accessed by an application during execution. This is implemented by having references passed to an application call transaction, which denote the on-chain data the application could access. These references are passed as reference arrays to the call and they may contain the following types of references: Accounts, Assets (ASA), Applications and Boxes.

When a smart contract developer wants to expose functions (or methods) he can use the Application Binary Interface (ABI) specification defined in Algorand Requests for Comment (ARC) 4 [37]. This way clients can easily interact with the application. Adhering to the ABI specification is not mandatory and many contracts are written without it. Method parameters can have the following types:

uintN An N-bit unsigned integer, where 8 <= N <= 512 and N % 8 = 0

byte An alias for uint8

bool A boolean value that is restricted to either 0 or 1. When encoded, up to 8 consecutive bool values will be packed into a single byte

ufixedNxM An N-bit unsigned fixed-point decimal number with precision M, where 8 <= N <= 512, N % 8 = 0, and 0 < M <= 160, which denotes a value v as v / (10̂M)

type[N] A fixed-length array of length N, where N >= 0. type can be any other type

address Used to represent a 32-byte Algorand address. This is equivalent to byte[32]

type[] A variable-length array. type can be any other type

string A variable-length byte array (byte[]) assumed to contain UTF-8 encoded content (T1,T2,...,TN) A tuple of the types T1, T2, . . . , TN, N >= 0

reference A uint8 parameter that points to an account, asset or application in the references array.

transaction A parameter that points to a transaction in a *transaction group*.

In contrast to Ethereum, application calls on Algorand are not marked as payable or asset transfer etc. To send a payment or an asset transfer with an application call Algorand makes use of its atomic transfers feature. When a client sees that a payment is need for a method then he would need to add the payment and the application call

in an atomic transfer. This way the payment and the application call will be executed together and the payment can be evaluated by the contract. These transactions can also be noted in the ABI as `pay` for payments, `acfg` for asset config/create, `axfer` for asset transfer, `afrz` for asset freeze and `keyreg` for key registration.

Algorand Smart Contract Security

Smart contracts on Algorand have their own set of common vulnerabilities. Most of the previously mentioned vulnerabilities (section 2.3) are not applicable to Algorand smart contracts for the following reasons:

- **Reentrancy** - Application calls and payment/asset transfers are different types of transactions in Algorand. Payment and asset transfers do not trigger any code execution. Also, an application cannot make a call to itself directly or indirectly. This means that reentrancy is not possible.
- **Integer Overflow and Underflow** - All overflows and underflows halt the execution and fail the current transaction.
- **Front-running** - The block proposer in Algorand is chosen randomly. The proposer cannot know beforehand whether he will be the next proposer. Therefore, the ability to front run is massively mitigated.
- **Time Dependence** - Although Algorand smart contracts can use a timestamp variable, it is very difficult to manipulate it. Similar to front-running, the block proposer cannot know beforehand whether he will be the next proposer.

From the previous list of vulnerabilities, only **Resource Limit Vulnerabilities** and **Simple Logical Errors** are applicable to Algorand smart contracts. The following list of common Algorand specific vulnerabilities that are commonly encountered [38, 15]:

- **Rekeying** - The `rekey-to` field is not checked by the smart contract. An attacker could use this to take control of the account associated with the smart contract.
- **Unchecked Group Size** - An attacker can create correct transactions at the particular indices. Then, he can populate the rest of the indices with malicious transactions. When the group size is not checked, the malicious transactions will be executed together with the correct transactions.

- **Uncontrolled update/delete operations** - Update and delete calls can be sent by anyone, and whether the transaction is approved depends on the program logic. Lack of appropriate checks would allow attackers to update the application logic or delete the application.
- **Asset Id Check** - Not checking the asset id, might allow an attacker to transfer assets that are not intended to be transferred.
- **Unchecked receiver** - As we mentioned, a payment or an asset transfer can be grouped together with the application call. The application call and the transfer even though grouped together, have no relation to each other. Developers (for example coming from Ethereum) might assume that the receiver of the transfer is automatically the account of the current application. An attacker could use this to his advantage.

These and other similar types of vulnerabilities are present in Algorand smart contracts. The commonality between these vulnerabilities is that they are very simple in nature. They are not caused by the complexity of the smart contract, but rather by the lack of proper checks. This simplicity of the common vulnerabilities and how it affected our design choices will be discussed in Chapter 3.

2.5. Fuzzing

The concept of fuzzing can be traced back to the 1980s when Professor Barton Miller at the University of Wisconsin experimented with sending random strings to UNIX utility programs [39]. Out of the programs tested 25% to 33% crashed, which at the time was a surprising result. From this work, the first fuzzing tool was built to allow other researchers to conduct similar experiments.

The main idea behind fuzzing is providing random input for the program under test and monitoring its behavior. Algorithm 1 shows a generic fuzzing algorithm, which is used in when modeling fuzzers [6].

Algorithm 1 Generic fuzzing algorithm

Input: $conf, t_{limit}$
Output: $\mathbb{R} - results$
$\mathbb{R} \leftarrow \emptyset$

while $t_{elapsed} < t_{limit} \wedge$ **Continue**$(result)$ **do**
  $input \leftarrow$ **InputGen**$(conf)$
  $result, execinfo \leftarrow$ **InputEval**$(input, O_{bug})$
  $conf \leftarrow$ **ConfUpdate**$(conf, result, execinfo)$
  $\mathbb{R} \leftarrow \mathbb{R} \cup \{result\}$
end while

The fuzzing process is usually represented by the following steps:

1. Input is generated given a fuzzing configuration. **InputGen** is the function that generates the input.
2. The generated input is evaluated against the program, given a certain bug oracle. **InputEval** is the function that evaluates the input.
3. The evaluation results are used to improve the fuzzing configuration. This is done by the **ConfUpdate** function.
4. The process is repeated until the time limit is reached or the **Continue** function decides based on the results that the process should stop.

Because there is a time limit on the fuzzing process, most fuzzers want to prioritize the input that is most likely to trigger a bug. This is usually done by using different metrics to decide which input is more interesting in this regard. One of the most popular metrics is code coverage. The idea is that the input that covers the most code is more likely to trigger a bug. In such fuzzers, interesting inputs are saved and prioritized based on the rareness of their coverage path. This measure of how interesting an input is, is often called the *energy* of the input. To facilitate this, the architecture of a fuzzer can be split into the following parts [40]:

- **Fuzzer** - Runner of the main fuzzing loop.
- **Schedule** - Decides which of inputs from the population of interesting inputs should be used by the fuzzer in the next iteration.

- **Input Generator** - Generates the input for the fuzzing process. There are many different types of input generators, the most popular ones being mutation-based, where old inputs are slightly changed to create new inputs. In Figure 2.2 we can see a Mutator as the chosen input generator.
- **Seed** - This represents an interesting input that contains that data, the energy and other information (such as coverage path) about the particular input.

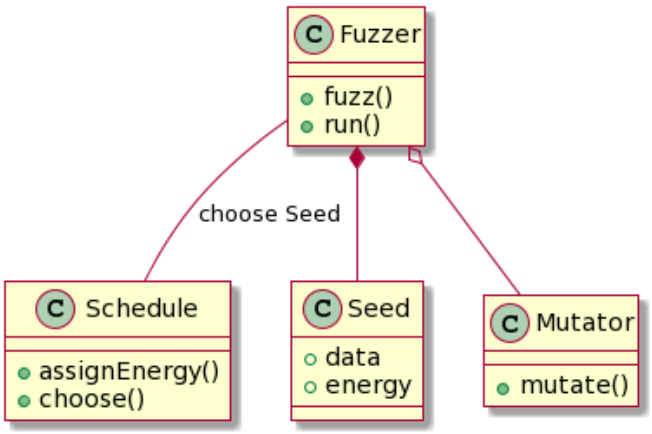


Figure 2.2.: Generic architecture of a fuzzer

2.6. Related Work

In this section we will explore the existing work in the field of smart contract fuzzing. Next, we review existing tools that aim to improve the security of Algorand smart contracts. Finally, we present the research gap that our work aims to fill.

Smart Contract Fuzzing

A number of fuzzers have been developed for smart contracts. The vast majority of them target Ethereum smart contracts. One of the most prominent fuzzers for Ethereum smart contracts is Echidna. Developed by the company Trail of Bits, Echidna is a property-based coverage-guided fuzzer that tests the contract's invariants using random transactions. Echidna has been employed in numerous security audits and has proven to be effective in uncovering bugs in Ethereum contracts [10, 11]. It makes

use of an EVM implementation written in Haskell called *hevm* [41], which allows it to run the contracts without the need of a private Ethereum network. *ContractFuzzer* was introduced in 2018, it stands as one of the earliest examples of fuzzing applied to smart contracts [42]. The fuzzer uses static analysis in combination with fuzzing to generate inputs that are more likely to trigger bugs. Different from Echidna, it uses a private Ethereum network to run the contracts. One big contribution of this work is one of the first introductions of specific bug oracles for Ethereum smart contracts. *Harvey* is one of the few closed sourced fuzzing engines [43]. It is among the most widely used fuzzers in the industry. The company Consensys has used it to build their smart contract fuzzing platform called Diligence Fuzzing [44]. There are several other fuzzers like *HFContractFuzzer* [45], *EOSFuzzer* [46], and *AntFuzzer* [47]. These fuzzers target other smart contract platforms like EOS [48] and Hyperledger Fabric [49].

Algorand Security Tooling

Algorand, while not as extensively explored as Ethereum in terms of security tooling, has seen efforts directed towards enhancing the security of its smart contracts. Some of the most notable tools are:

- **Tealer**: This is a static analysis tool that checks for common vulnerabilities in Algorand smart contracts [14]. Most of the checks are based on the vulnerabilities that we mentioned in section 2.4. As the vulnerabilities are simple in nature, the tool is capable of detecting most of them.
- **Panda**: It is another static analysis tool for Algorand contracts [15]. Different from Tealer, the user can define custom rules to check for vulnerabilities, aside from the ones that are already implemented. The tool has found many vulnerabilities in real world projects which have been acknowledged by the respective teams.
- **AlgoBuilder**: A framework for building dApps for Algorand [16]. Included with the framework is a testing tool that allows the developer to write unit and integration tests for their smart contracts.

Research Gap

Despite the efforts and tools available for ensuring the security of Algorand smart contracts, there are challenges with the tools currently in use. For instance, static analysis tools are limited in their ability to detect vulnerabilities, as they often overlook

vulnerabilities or produce false positives. This underscores the need for a complementary dynamic analysis tool such as a fuzzer. While Ethereum has seen a multitude of fuzzers, Algorand lacks a dedicated fuzzer tailored for its unique smart contract architecture and the TEAL scripting language. In the GitHub repository of Algorand, there is a fuzzer called *TealFuzz*, but the project is currently abandoned, with 17 commits in total and the last commit being made in 2020 [50].

As Algorand continues to gain traction, the complexity of its smart contracts will increase. The security of these contracts will need to be handled from all angles. Therefore, a dedicated fuzzer for Algorand smart contracts is a necessary addition to the security tooling of the blockchain platform.

3. Methodology / AlgoFuzz

The primary objective of this research is to design and develop AlgoFuzz, a fuzzer tailored for Algorand smart contracts. This chapter will detail the methodology employed, breaking down each step and rationale.

3.1. Fuzzing Approach

Input Generation

When making an application call on Algorand, the data that is passed can be completely arbitrary. The smart contract developer can decide on the format of the data and how it should be interpreted. This means that the general approach of generating random inputs, could be suitable for such cases. However, generating inputs in a completely random fashion is not very effective because the probability of generating a valid input is very low. The vast majority of inputs will be rejected by the application. For this reason, our fuzzers considers only contracts that adhere to the official Algorand ABI conventions (mentioned in 2.4). Our input generators will be aware of the input structure of the contract. This provides us with the different functions that are available in the smart contract and the types of the arguments that they take. From this information, comes the first part of the fuzzed input, the second part is the account that is used to call the smart contract. Our fuzzing input would then be the following tuple: `(function, arguments, account)`. To generate the inputs for our fuzzer we will be using a mutation-based approach. This approach is a good starting point since it is simple to implement, has been shown to be effective in previous works [40] and it can be extended in the future to include more sophisticated techniques. For now, the inputs will be seeded with a minimal value according to their type, such as an empty string.

Fault Detection

Based on our research on Algorand smart contracts, we decided to develop a property-based fuzzer for Algorand. As discussed in 2.4, detecting errors that commonly happen in Algorand smart contracts can be better done by a static analyzer. For the evaluation of the input, the user can define properties that should hold for the contract. One such property could be that the balance of the contract must always be below a certain threshold. Alternatively, they can run the fuzzer in *Assertion Mode* which will check for all the possible assertion failures on application calls. Assertion mode may be used when the user has a contract which contains assertions that should never fail. The properties that the user can define will be dependent on the state of the contract. With the state, the user can define arbitrary invariants which should always hold (return `true`). Due to limitations of the current version of Algorand nodes we will only be able to retrieve the global and local state of an application. The methods that manipulate the box state of the application are currently not implemented for `dryrun` calls. Dryrun calls are used to simulate the execution of a transaction without actually executing it. They are important for our fuzzer since they allow us to retrieve the coverage information of an application call, which is used as a guiding metric for the fuzzing process (discussed in 3.3).

Granularity of Analysis

For the fuzzer to be as general as possible, we decided to make it work with any contract written in TEAL without considering the higher level tool (such as PyTeal or TEALScript) used to write the contract. Although the fuzzer will have access to the TEAL code of the contract and the ABI, it will not be able to leverage white box fuzzing techniques. We decided this was outside the scope of this research due to the complexity of implementing such a fuzzer. Such a fuzzer would need to be able to parse the TEAL code and understand the semantics of the different opcodes.

For our project, we decided to focus on a grey box approach. The inputs generated by the fuzzer will be guided by different metrics that we gather during the process. Concretely, when an input covers more code than previous inputs or produces a new state transition, it is deemed interesting and saved to be mutated later. The first metric, code coverage, is the most common metric used to guide the fuzzing process. An advantage of using coverage is that it gives us a metric that can be used to compare our fuzzer to other fuzzers. The reason we also consider state transitions is because our

fuzzer relies on property tests on the state of the contract to detect bugs. Therefore, it makes sense to have a fuzzer which generates inputs that produce novel states.

Stopping Condition

Choosing the stopping condition for a fuzzer in general is still an open research problem. The stopping condition of AlgoFuzz can be configured in multiple ways. Similar to Algorithm 1, the fuzzer can be configured to run for a maximum amount of time. Alternatively, the maximum number of calls executed by the fuzzer can be specified. In both cases, the fuzzer will be stopped prematurely if an input breaks a property test or if an assertion failure is detected. This is slightly different from the approach most fuzzers take, where the fuzzer runs for the whole duration and gathers all possible bugs during the process. The reason for this is that we want to see if there is a bug in the contract, opposed to how many bugs there are. Smart contracts are usually small programs [51], so they are not expected to have many bugs.

3.2. AlgoFuzz Design

Execution Environment

Contract. The smart contracts being fuzzed will be deployed on a local Algorand network. This network usually comes with the official AlgoKit CLI tool [52]. The setup for such a network is straightforward and can be done with a single command. The local network starts three docker containers to simulate the Algorand blockchain with a REST API for users to interact with the local network outside the container.

We also looked at different interpreters for TEAL that simulate the AVM such as teal-interpreter [53] and the AlgoBuilder Runtime [54]. We decided against using these tools for the following reasons:

- They are not official tools and are not maintained by the Algorand Foundation.
- They are not as well documented as the official tools.
- Executing the smart contracts on the local network is most similar to how they would be executed on the Algorand network.
- teal-interpreter has had no commits in more than one year, supports only up to TEAL version 5 (current version is 8) and does not support all opcodes.

Fuzzer. The fuzzer is implemented as a standalone Python application with a command line interface. It connects to the local network through the REST API by using the Algorand Python SDK [55]. Most Algorand contracts are written in PyTeal, which is a Python library that offers a higher level abstraction over TEAL. For this reason, we decided to implement the fuzzer in Python as well which simplifies our development process. Aside from the lower mental overhead of using the same language, with Python we can compile our PyTeal contracts and then immediately use them in our fuzzer. Fuzzers usually rely on generating a large amount of inputs and therefore need a very performant language. Although Python is not the fastest language, we observed by profiling AlgoFuzz that most of the time was spent on the API calls to the local network (over 90%).

Architecture

The architecture of our solution is shown in Figure 3.1. We extended the generic architecture in Figure 2.2 by adding multiple ways to guide the fuzzing process through our different drivers which are passed on to the `ContractFuzzer`. Aside from that, we use mutations for our input generation. Since we have to support multiple different data types, the base Mutator is shown as an `Interface` which will be implemented by specific Mutator classes. To handle operations connected to the local network, we created `FuzzAppClient` which is a wrapper around the `ApplicationClient` from the AlgoKit. It encapsulates all the logic for making calls to the local network and retrieving the results. In Section 3.3 we will go over the implementation details of the different modules.

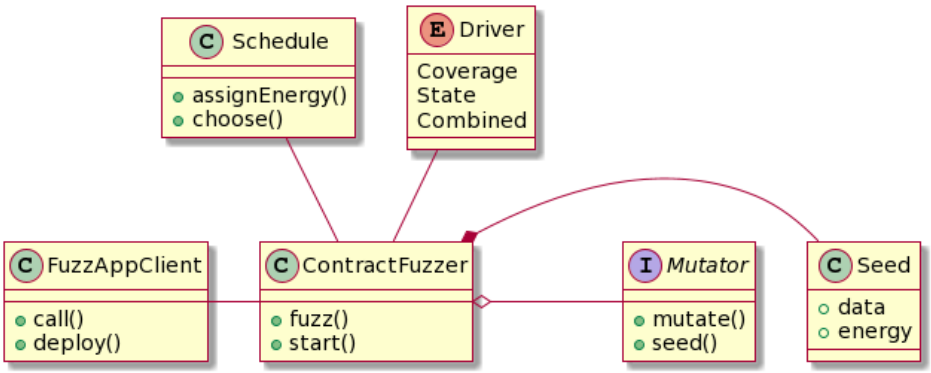


Figure 3.1.: Architecture of AlgoFuzz

3.3. Implementation

FuzzAppClient

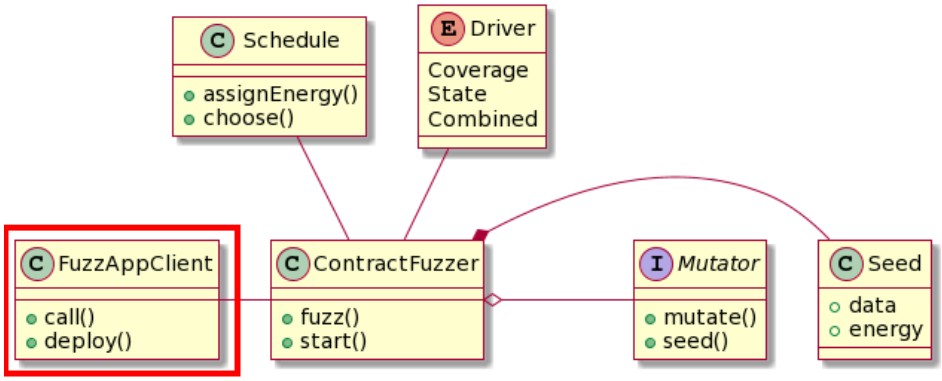


Figure 3.2.: AlgoFuzz architecture with FuzzAppClient highlighted.

The first thing we had to do was find a way to interact with the local network. In the beginning, we used the Algorand Python SDK directly to make calls to the local network. This worked fine for the most part, but it required us to write a lot of code for generic operations when we wanted to interact with an application. For example, to make an application call we had to create a transaction, sign it, send it to the network, wait for the transaction to be confirmed, and then retrieve the result. Later we found out that Algokit provides a standard client for applications, called `ApplicationClient`, which encapsulates most of these operations, and it also works with TEAL contracts. To make the client more suitable for our use case, we created a wrapper around it called `FuzzAppClient`. This wrapper adds the following functionality:

- Gathering coverage information from every application call. This is done by making a `dryrun` call before the actual application call.
- Preparing application call arguments such as payments, by creating the corresponding transactions.
- Changing the account that is used to make the application call.
- Opting in to the application all the accounts that are used for the fuzzing process.
- Disassembling the TEAL code of the contract and retrieving the total number of lines.

Mutators

In this section we will go through the mutators and the heuristics that we implemented for our fuzzer. A mutator has two important methods:

- `mutate`: This method takes an input and mutates it. Internally, a mutator may have multiple mutation strategies that it uses to mutate the input.
- `seed`: This method seeds the input with a value. Currently, we have chosen simple initial values for all different types.

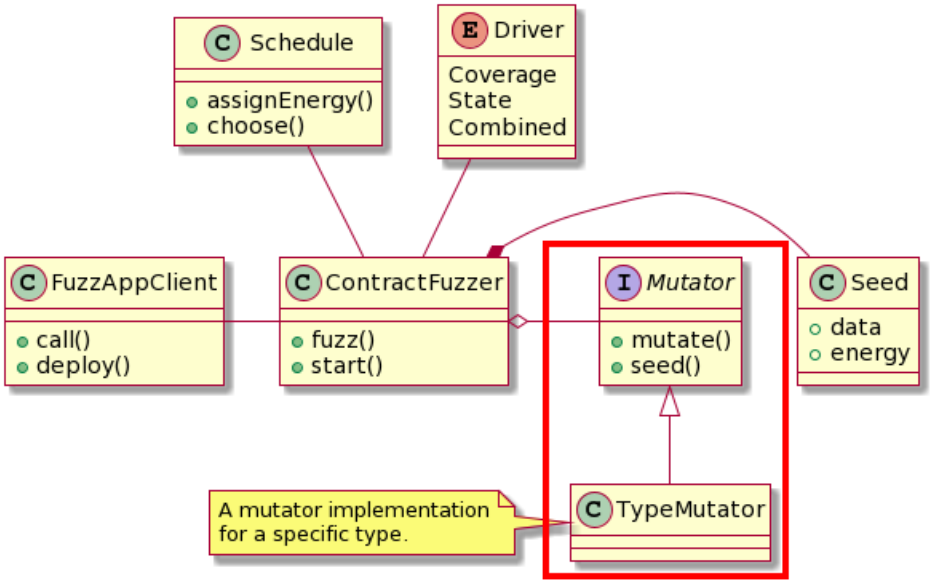


Figure 3.3.: AlgoFuzz architecture with mutator highlighted.

Boolean. This mutator generates random boolean values. The values are all seeded with `False`. The mutation strategy is simply to flip the value of the boolean.

Uint, Ufixed and Byte. The Uint mutator generates random unsigned integer values. It is initialized with the size of the integer, from 8 to 256 bits. The seed value used is 0, since it is the smallest possible value and also a commonly used value. The mutation function chooses randomly from the following mutation strategies are used:

- add - Adds a random value without overflowing.

- subtract - Subtracts a random value without underflowing.
- multiply - Multiplies the value with a random value without overflowing.
- divide - Divides the value with a random value without underflowing.
- random bit flip - Flips a random bit in the value.
- bitwise and - Performs a bitwise and with a random value.
- bitwise or - Performs a bitwise or with a random value.
- bitwise xor - Performs a bitwise xor with a random value.

The effectiveness of these strategies in fuzzing has been shown in previous works [56] and we expect them to be effective in our case as well.

Ufixed and Byte mutators are special cases of the Uint mutator. The Ufixed mutator is initialized with the size of the integer part and the size of the fractional part. When mutating the value is first converted to an unsigned integer, mutated and then converted back to a fixed point number by dividing it with the fractional part size. The Byte mutator is an alias for the Uint mutator with a size of 8 bits.

String, Array Dynamic and Array Static. The String mutator and the dynamic length Array mutator work similarly. They are seeded with an empty string or array. Then one of the following mutation strategies is randomly chosen:

- add - Adds a random value to the string or array at a random position.
- remove - Removes a random value from the string or array at a random position if the string or array is not empty.
- flip - Replaces a random value in the string or array with a random value.

The difference between the two is that the array mutator also initializes a mutator for the type of the array. This mutator is used when generating the random values to be added by mutating the seed or when flipping a value by mutating the value at the corresponding position. The static length Array mutator is identical to the dynamic length Array mutator, except that it can only use the flip element mutation strategy. Therefore, not changing the size of the array.

Tuple. For the tuple mutator, we initialize a mutator according to each type in the tuple. When mutating the tuple, we mutate each element of the tuple with the corresponding mutator. The seed value is a tuple of the seed values of the mutators.

Account. The account mutator is needed not only to generate an account as an argument for an application call but also to generate the account which will be executing the call. This mutator is initialized with a list of three funded accounts. Each account having a balance of 200 Algos. The seed value is the first account in the list. When the mutation method is called, we either return the current account passed to the mutator or we choose a random account from the list of funded accounts. This means that there is a 66% chance that the account will stay the same and a 33% chance that it will change. The reason for this is that we do not want the account to change too often. This ensures that a number of calls will be executed from the current before switching to another account.

Payment. The payment mutator is used to generate payment objects for method calls which require a payment. The payment object contains only the amount of Algos to be transferred. These objects are then transformed into a payment transaction and grouped with the application call. The recipient of the payment is always the account of the application being fuzzed. Since we are using a local network, we cannot send arbitrary amounts of Algos as there is a limit on the amount of Algos available. The amount of Algos sent will be a random value between 0 and 100 Algos, which is around the average transaction on Algorand [57]. In the future this could be a configurable value for the fuzzer. When one of the accounts has less than 200 Algos, we fund the account with an additional 1 million Algos. This way our transaction does not fail due to insufficient funds.

Method Mutator. The method mutator is initialized by passing an ABI method. Similar to the tuple mutator, we initialize a mutator for each argument of the method. The mutating strategy differs from the tuple mutator. We first generate a random number from 1 to the number of arguments of the method which will be the number of arguments that we mutate. Then we use this number to sample the indices of the arguments that we will mutate. For the sampled indices, we mutate the argument with the respective mutator. The seed of this mutator is a tuple of the seeds of the argument mutators. When generating our seeds for the whole contract we combine the method seed with each account. Meaning that we have a seed for each method and account combination.

Contract Fuzzer

For most smart contracts, the code of different methods is relatively independent to each other. Meaning the vast majority of methods are small, self-contained, and they rarely call common subroutines. From this point of view each method of the contract can be seen as a separate program. This means that we can fuzz each method separately and then combine the results. We call this approach *partial fuzzing* (similar to a partial derivative in math). The second approach that we implemented is *total fuzzing*. In this case, the fuzzer contains all necessary fuzzing parameters to fuzz the entire contract. In Figure 3.4 we can see the updated architecture with the two types of fuzzers.

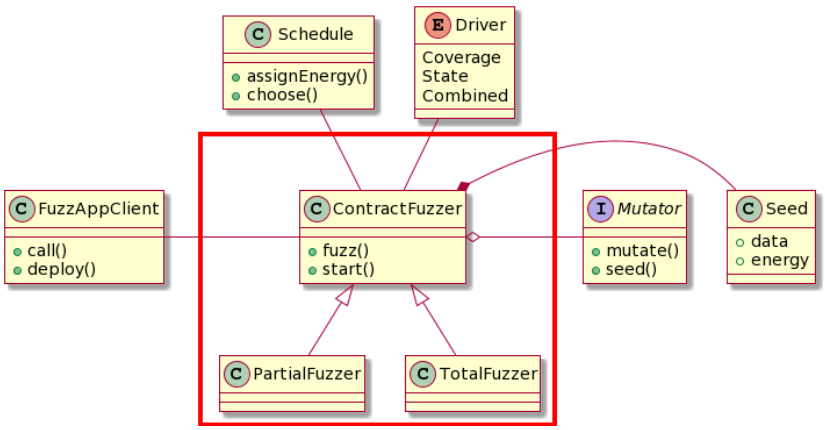


Figure 3.4.: AlgoFuzz architecture with two types of fuzzers.

Partial Fuzzer. During fuzzing, the partial fuzzer creates only one application and a separate fuzzer for each method. These *method fuzzers* each have their own population of interesting inputs, power schedules, seeds, and other fuzzing parameters. The population of inputs in this case being a list of tuples of the form `(arguments, account)`. On each iteration the partial fuzzer randomly chooses a method and runs the corresponding method fuzzer. All methods have the same probability of being chosen regardless of the amount of interesting inputs that are found for them. Although this approach seems naive, it has been shown to be effective in terms of code coverage from our experiments. The main advantage of this approach is that it is very easy to implement and it generates new inputs faster. The performance gain comes from the fact that the schedulers of the different method fuzzers have fewer inputs to choose from. Another subtle advantage of the partial fuzzer is that methods which do not

produce any interesting input during the seeding stage will not be completely ignored in the future fuzzing process.

Total Fuzzer. Different from the method fuzzers in the partial fuzzer, the population of inputs is a list of tuples of the form `(function, arguments, account)`. This means that functions that produce more interesting inputs will be fuzzed more often which is the main advantage compared to the partial fuzzer. The main disadvantage is that methods which do not produce any interesting input during the seeding stage will be consequently completely ignored. To avoid this problem, we introduced a breakout mechanism through a coefficient passed to the constructor of the fuzzer. This coefficient is used to determine the probability of uniformly randomly choosing one of the seeds and producing a new input by mutating it. We applied this breakout mechanism to the partial fuzzer as well.

Drivers

Most fuzzers use different metrics to determine which inputs are interesting and which are not. We call these metrics the *drivers* of the fuzzer. In this section, we will discuss the different drivers that we chose for our fuzzer.

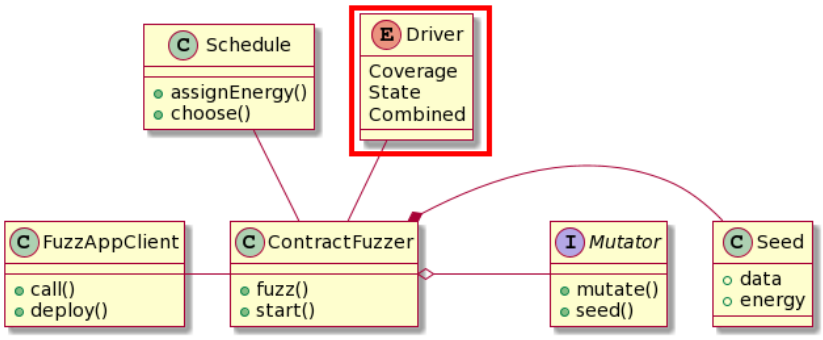


Figure 3.5.: AlgoFuzz architecture with driver highlighted.

Coverage Driver. As mentioned most fuzzers use some sort of coverage information to guide the fuzzing process. This is also the case for our fuzzer. The coverage information is retrieved by running the application call in `dryrun` mode. What we retrieve from the `dryrun` call tells us which lines of the code were executed and in what order. The coverage information of a call is processed by creating a set of all the lines that were

executed. This set is called a coverage path. If an input produces a new coverage path, it is considered interesting. The reason we use coverage paths instead of coverage lines is that the latter can be very large. Also, if we had to compare two lists of lines they may contain the same lines but in a different order. An example for this would be a loop.

State Driver. Since our fuzzer relies on property tests on the state of the contract to detect bugs, it makes sense to have a fuzzer which optimizes for generating inputs that produce different states. To use the state as a driver, we retrieve the global state of the application and the local states of the accounts in the account mutator. We then hash this information by creating a deterministic json representation of the state and then using the `MD5` hash function. This is necessary since the state can be very large and storing it for each input could be very expensive. One way of using this information to identify interesting inputs is by detecting if it produced a new state. This works fine, but we decided to consider inputs that trigger transitions between states as interesting. The reason for this is that in the future these transitions can be used to find the input sequence that produces a bug. This can be done by going through the transition graph and finding the shortest path between the initial state and the state that produces the bug. This process is similar to *shrinking*, where failing test cases are simplified to find the minimal reproducible case. The shrinking process is not implemented in our fuzzer, but it is a possible extension.

Combined Driver. The combined driver is a combination of the coverage and state driver. When we use this driver, input is considered interesting if it produces a new transition or a new coverage path. We combined the drivers by introducing a coefficient that denotes the weight of the state driver. This coefficient is then passed to the schedule, which is discussed in the next section.

Schedule

The schedule of AlgoFuzz is implemented through an exponential power schedule similar to the one proposed by AFLFast [58]. This schedule assigns high energy to inputs that traverse paths or transitions that have been less frequently traversed. How often a path or transition has been traversed is stored in dictionaries, one for the path frequency and one for the transition frequency. Every time an input is executed the frequency of the path and transition that it traversed is incremented. The keys of the dictionaries are the hashes of the paths and transitions. As mentioned above, these

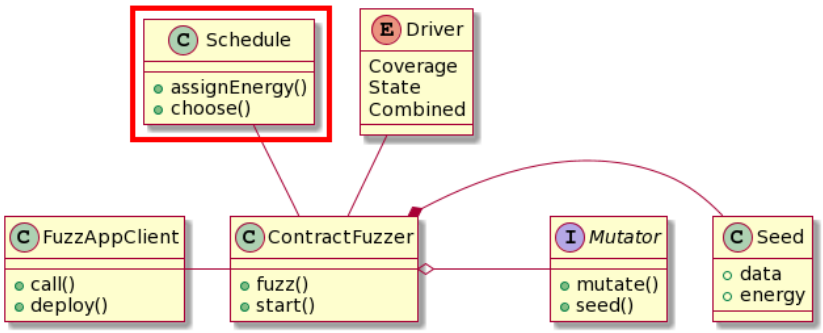


Figure 3.6.: AlgoFuzz architecture with schedule highlighted.

frequencies are then used to first calculate the weighted frequency given the coefficient of the state. The formula for the weighted frequency is the following:

$$w(s) = w_s * t(s) + w_c * p(s)$$

s	seed
w_s	state weight
w_c	coverage weight (1 - w_s)
$t(s)$	frequency of transition traversed by s
$p(s)$	frequency of path traversed by s
$w(s)$	weighted frequency of s

Then the energy is calculated with the following formula:

$$e(s) = \frac{1}{w(s)^{\alpha}}$$

α	schedule exponent
$e(s)$	energy of the seed

The schedule exponent is a parameter that can be configured for the schedule. It determines by how much we prioritize less traversed paths and transitions. For AlgoFuzz this value is hard coded to 5, this way we emphasize newer inputs more aggressively. In the future, this could be a configurable value for the fuzzer or it could be dynamically changed during the fuzzing process.

After calculating the energy of each input, the schedule uses it to determine the probability of choosing an input. The probability of an input being chosen is the normalized energy, which is calculated by dividing the energy of the input by the sum of all energies.

3.4. Limitations

Despite the advantages, using the technologies and approaches that we have chosen comes with some limitations. The first one being that the performance of our fuzzer will be limited by the performance of the local network. The local network is relatively slow since it simulates all Algorand blockchain operations, such as adding and verifying blocks. Due to this, we also need to fund the accounts that we use for the fuzzing process which requires a transaction from a dispenser account to the account being funded. Parallelizing the fuzzing process would also not work. Even though the local network can take multiple transactions at the same time, it will only execute them sequentially. Aside from this, making requests back and forth to the local network is slower than having an interpreter in the same process. Also, to retrieve the coverage information of an application call, we need to make two requests to the local network (`dryrun` and the actual transaction).

The other limitations are of functional nature. As mentioned we cannot access the box state of the application, because it is not supported by the `dryrun` call. This means that contracts that rely on dynamic storage are not supported or impractical to replicate for our fuzzer.

Our tool will initially not be supporting all argument types from the ABI, but it will be supporting the most common ones. The most important types left out will be:

- `address`: This type is used to represent an address on the Algorand network. Giving random addresses in this case is not a good idea since the address is a 58 byte array that is encoded in base32. This means that the probability of generating a valid address is very low. For this reason, we decided to leave this type out of the initial implementation.
- `reference types`: Aside from the account reference type, all other reference types will not be supported since they are difficult to generate. For example, working with assets requires the user to opt-in to the asset as well as the application being fuzzed. Since this feature is not currently supported also generating an asset reference is not possible.

Another limitation of our tool is that it expects opt-in calls to be empty. In some Algorand contracts, opt-in calls may be connected to a certain function call that requires specific arguments. Currently, it is not possible to infer from the ABI of the contract with which function an opt-in call is connected to. Therefore, we decided to leave this feature out of the initial implementation and we opt in all required accounts with empty application calls. Similarly to the opt-in calls, all other special calls are not executed by our tool. The fuzzer only executes noop applications calls. In the future, we could extend our tool to also randomly execute special calls.

3.5. Important Runtime information

In this section, we will discuss the different metrics that we used to evaluate our fuzzer.

Number of calls executed Shows us how efficient the fuzzer is at generating inputs and calling the application. Since fuzzing is a random process, the number of calls executed needs to be very large to get meaningful inputs.

Number/Ratio of rejected calls Any call that is rejected by the application is not interesting. The ratio is the number of rejected calls divided by the total number of calls. The lower the ratio, the better. Fuzzers that have a low ratio of rejected calls have a higher probability of generating interesting inputs and finding the contract.

Number/Ratio of lines covered The number of unique lines covered is the number of lines that were executed during the fuzzing process. The ratio is the number of unique lines covered divided by the total number of lines in the TEAL approval program of the contract. This is a metric that is used by most fuzzers.

Number of unique paths Every set of lines covered produced by a call is a coverage path. This gives us a metric of how many different paths were executed during the fuzzing process. In general, the more paths are covered, the better. Although, a higher number of paths does not necessarily mean that more lines were covered. This is because paths can have different lengths and a path can be a subset of another path.

Number of unique state transition We record all unique state transitions that were produced by the fuzzing process. The more transitions are produced, the closer the fuzzer can be to finding a state that would invalidate the property test.

3.6. Case Study: AlgoTether

Before we go into the evaluation of our fuzzer, we will go through a case study to determine which parameters work best for our fuzzer. The contract is called AlgoTether and it is a simple contract that allows users to mint and transfer tokens. AlgoTether is translated from the Solidity contract of the Tether Token [18], which is a stablecoin that is pegged to the US dollar.

The reason we decided to translate this contract is because there are no Algorand contracts that use the official ABI and do not rely on box storage. This makes it hard to find contracts that are long enough to be interesting for our evaluation.

Contract Features

This contract represents an ERC20 token [59]. Each account has a balance of tokens that can be transferred to other accounts. An account can also allow other accounts to spend tokens on its behalf. In the Algorand version of the contract, we represent balances as local state in the account that owns the tokens. For allowances, we store in the local state of the account that owns the tokens, a mapping from the account that is allowed to spend tokens to the amount of tokens that it is allowed to spend. The key for the allowances is `allowance_<spender>` where `spender` is the address of the account that is allowed to spend tokens. Since we are using local state we need to declare it before deploying how much space we will need. In our case, we will need only 3 local state values for the account, since we have at maximum 3 accounts.

The contract is *ownable* which means there is an owner account that has special privileges and can perform certain operations that other accounts cannot. The owner of the contract is the account that deployed the contract and it can be changed only by the current owner. The contract is also *pausable*, meaning that the owner can pause the contract, which will prevent any transfers from happening.

The contract also has a blacklist feature, which allows the owner to add accounts to a blacklist. Accounts on the blacklist cannot transfer tokens. The blacklisted accounts are stored on the global state of the contract, with the key `blacklist_<acc>`.

Setup

For this case study, we tried all the different combinations of drivers and fuzzers. This gives us in total 6 different fuzzers: partial fuzzer with coverage driver, partial fuzzer with state driver, partial fuzzer with combined driver, total fuzzer with coverage driver, total fuzzer with state driver, and total fuzzer with combined driver. We also tested the total and partial fuzzer with the combined driver using different state coefficients (or the weight of the state driver). For this we used the weights from 0.1 to 0.9 with a step of 0.1. The weights of 0 and 1 are not included since that would be almost the same as using coverage or state driver respectively.

The case study was conducted on a local machine with an Intel Core i7-8550U CPU and 16GB of RAM. Each fuzzer was run 5 times with a timeout of 2 minutes and the averages of the runs were gathered. Since smart contracts are small programs, the fuzzer can achieve meaningful results for our case study in a short amount of time.

Results

First we ran the experiment with the Combined Driver using different coefficients (state weight) to find out which one works best. We focused on two key questions:

- Which coefficient leads to more code being covered?
- Which coefficient produces more unique state transitions?

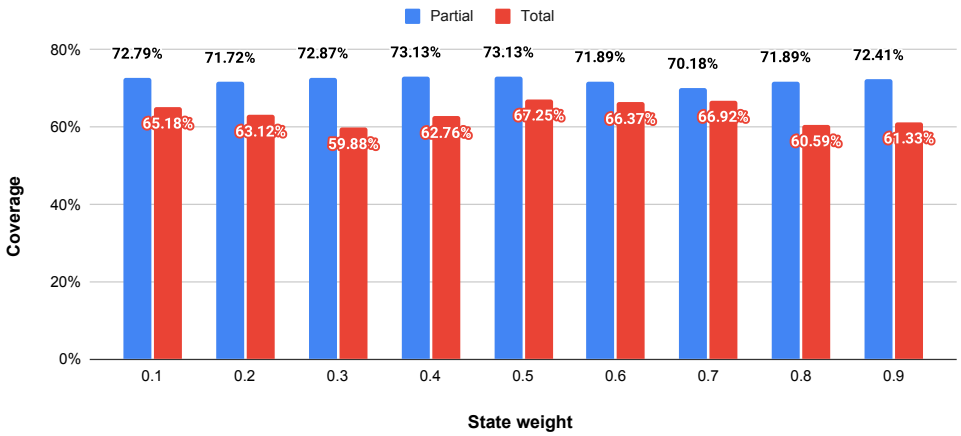


Figure 3.7.: Coverage results with different coefficients.

To answer the first question we just looked at the percentage of lines covered which is shown in Figure 3.7. From which we can clearly see that the highest values are concentrated at the coefficient of 0.5.

For the second question we looked at the number of unique state transitions, which is shown in Figure 3.8. Here it is less clear which coefficient works best. For the partial fuzzer, the number of transitions goes down as the state coefficient increases. This seems counterintuitive at first, but it makes sense when consider that the number of calls executed also goes down. When we calculate the ratio of transitions to calls executed, we see that it is mostly staying the same throughout the different coefficients (at around 22% for the partial fuzzer and 11% for the total fuzzer).

In the case of state discovery it is difficult to say which coefficient works best. Therefore, based on the coverage results we chose the coefficient of 0.5 for the combined driver.

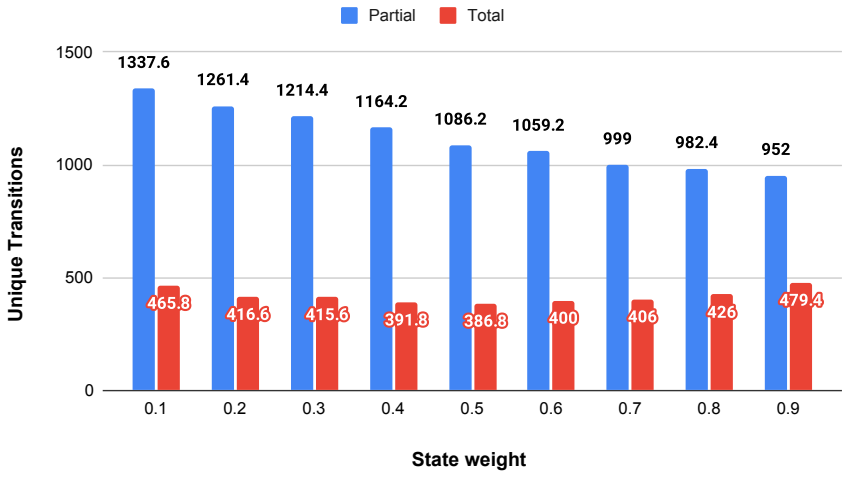


Figure 3.8.: State transitions with different coefficients.

With the coefficients chosen, we ran the experiment again to compare the different fuzzers. The results for rejected calls and coverage are shown in Figure 3.9.

From the results we can see that the total fuzzer with the state driver has the lowest ratio of rejected calls by far. This means that it is the most efficient at generating inputs that are accepted by the contract. Aside from this, we can see that the partial fuzzer

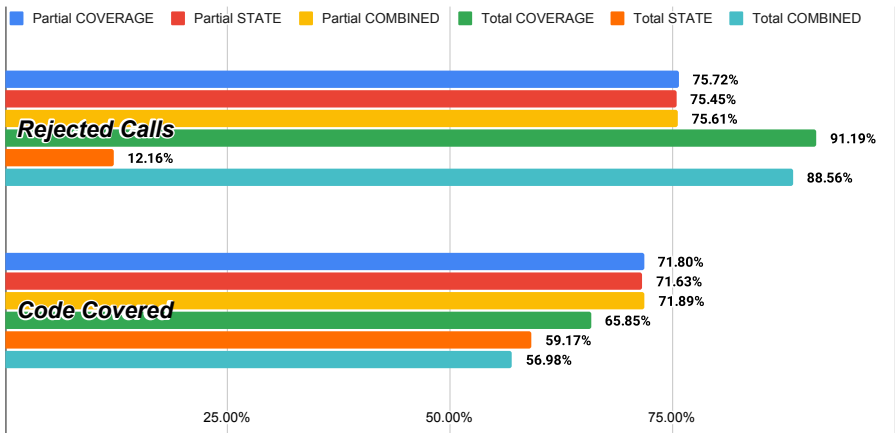
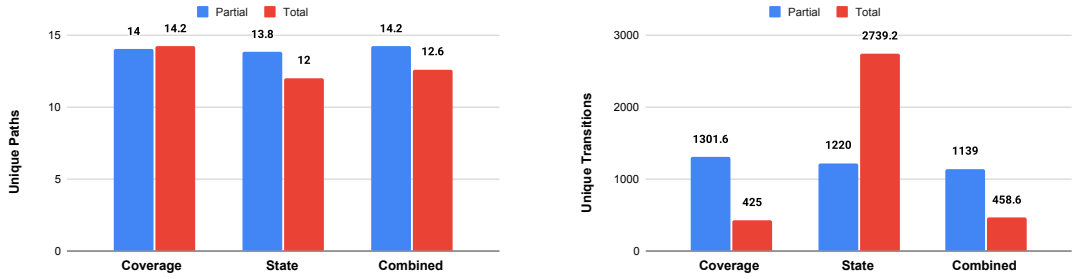


Figure 3.9.: State transitions with different coefficients.

has higher coverage than the total fuzzer across all drivers. And, as expected, for the total fuzzer the coverage driver has the highest coverage.

It was also interesting to see the amount of unique paths and state transitions that were covered by the different fuzzers, shown in Figure 3.10.



(a) Unique Paths.

(b) Unique Transitions.

Figure 3.10.: Unique coverage paths and state transitions discovered.

Different from the coverage results, the total fuzzer is not too far behind in terms of the amount of unique paths discovered. For the state transitions, we see the partial fuzzer performs more or less the same regardless of the driver, at around 1200 transitions. In contrast, the total fuzzer underperforms for the coverage and combined drivers at around 430 transitions. The total fuzzer with the state driver show outstanding results with 2739.2 transitions discovered on average. This goes hand in hand with the rejected calls results, since the state driver will most likely prioritize inputs which

slightly change the state of the contract and therefore are safer.

From these results we can make the following preliminary conclusions:

1. The total fuzzer with the state driver is the most efficient at generating inputs that are accepted by the contract.
2. The total fuzzer with the state driver is also the best at discovering new state transitions.
3. The partial fuzzer outperforms the total fuzzer in terms of coverage across all drivers.

4. Evaluation

Our original goal was to bring a first prototype fuzzer to the Algorand ecosystem of tools and for this reason we designed and implemented AlgoFuzz. In this chapter, we will evaluate AlgoFuzz to see if it fulfills its purpose as a fuzzer for Algorand smart contracts. The most straightforward way to find out if a fuzzer works is to see if it is able to find bugs, known and unknown. Looking at other smart contract fuzzers, such as ContractFuzzer or Harvey, we can see that the most common approach is to run the fuzzer on real-world contracts. After which, they would see if their fuzzer reported any bugs during the process. In our case, since we are dealing with a property-based fuzzer and we do not have a list of predefined bug oracles, we would need to know the contracts in detail to be able to write properties that would find bugs. Echidna, the property-based fuzzer for Ethereum, was also faced with the same problem during their evaluation. Instead of finding bugs, they decided to see if their fuzzer was able to reach certain assertion failures which were randomly added to the contracts.

With these considerations in mind we came up with two different questions that we wanted to answer for our evaluation:

RQ.1 How effective is AlgoFuzz in terms of covering new code?

Code coverage has been shown to correlate with the number of bugs found [17].

RQ.2 How effectively can AlgoFuzz discover the state space of the smart contract?

Since the property tests are defined on the state of the contract, the more state the fuzzer discovers the more likely it is to invalidate a property.

4.1. Setup

Chosen Contracts

For our evaluation, it was difficult to find real-world Algorand smart contracts which adhered to the limitations of our fuzzer namely using the official ABI, not using any

box state and not interacting with ASAs. For this reason we decided to write our own contracts in PyTeal based on already existing Solidity smart contracts which where used in previous works. This way we can leverage existing contracts, while still adhering to our limitations. The first set of contracts we chose were the twelve contracts used in the Echidna evaluation [10], namely a subset of the VeriSmart-benchmarks [60]. For our second set of contracts we chose two contract of a larger size than the ones selected before. The first one being the Tether stablecoin contract, presented in our case study in 3.6. The second one is a generic token exchange contract, where the user can buy and sell tokens for Algos given an exchange rate, as this is a common use case for smart contracts.

Evaluation Protocol

The metrics we gathered for our experiments were the ones mentioned in section 3.5. We ran each configuration of the fuzzer as we did in the case study, so six configurations in total (two different fuzzing strategies and three different fuzzing drivers). For the 12 small contracts we did 10 runs per configuration with a fuzzing time of 10 minutes each. As these contracts are small the fuzzer should be able to reach a saturation point in this time. In total the fuzzer ran for these contracts for 120 hours. The two larger contracts also ran for 10 times per configuration but with a fuzzing time of 30 minutes each, as these contracts will take longer to reach a saturation point. For the larger contracts the fuzzer ran in total for 60 hours. All in all the experiment ran for 180 hours. After each call to the target contract we recorded the metrics in a CSV file. The results that we show are the average of the 10 runs for each configuration.

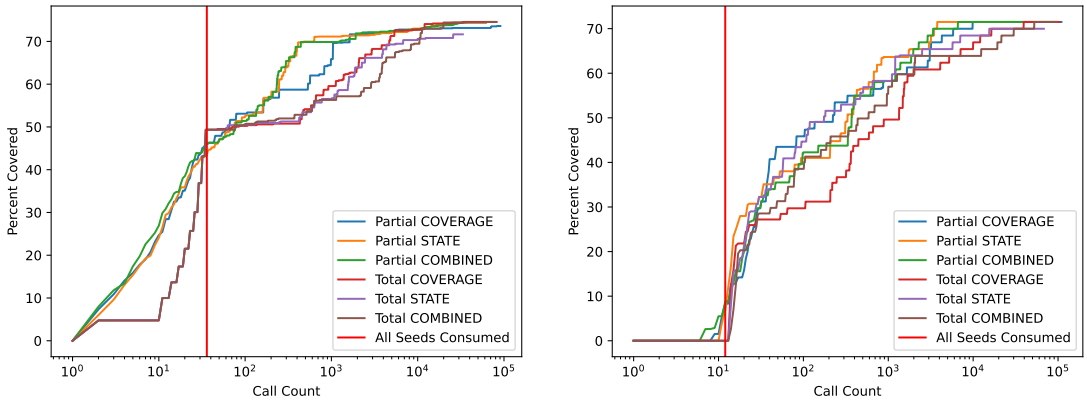
The experiments were run on a virtual machine hosted on the LRZ Compute Cloud [61]. The compute node uses Intel(R) Xeon(R) Gold 6148 CPUs at 2.40GHz. The virtual machine had 2 CPU cores, 9 GB of RAM and 30 GB of disk space. The operating system was Ubuntu 20.04 LTS. AlgoFuzz used the Python 3.10.12 interpreter, the Algorand node was version 3.15.1, and the Docker version was 24.0.6.

4.2. Results

In this section we will present the quantitative results of our evaluation. We also elaborate on how the data was selected and transformed to be able to present it in a meaningful way.

From the data we collected we excluded some runs which were uncharacteristically slow, assuming that they were statistical outliers caused by the virtual machine. The runs we excluded were the ones where the number of calls was much lower than the average for the same configuration. These were the first 3 runs on AlgoTether contract with the Total Fuzzer and the Coverage driver.

We wanted to aggregate the data of each configuration per call. One issue with this approach is that due to the random nature of the fuzzer, the number of calls per run was almost never the same. For all runs in a configuration, we calculated the average, median, minimum and maximum of each metric per call. For the averages we only considered the available data points, so if a run had no calls beyond a certain point we did not consider it for the average. This does not affect our values since most runs achieve a similar number of calls and by the end they have all reached a saturation point. This was only done for the graphs where we are showing the values per call. For the final values we took the run with the least number of calls and used that index to get the final values for all runs. In some graphs we will only be presenting the values of the larger contracts, since the smaller ones reach a saturation point very quickly and they do not change significantly after that. Aside from that we calculated the values for the last call of each run to see what the final state of the fuzzer was.



(a) AlgoTether

(b) ExchangeToken

Figure 4.1.: Average coverage of AlgoTether and ExchangeToken for all configurations.

Code Coverage

To answer **RQ.1** we will look at percentage of code coverage over the number of calls to the target contract. Other metrics such as lines covered or the number of unique paths covered were also considered but were discarded since they did not provide any additional information. The percentage of code coverage is a normalized metric making it is easier to compare the results of different contracts.

In Figure 4.1 we can see the percentage of code coverage over the number of calls for the two larger contracts. The x axis which represents the number of calls is shown in logarithmic scale. This is because coverage increases very quickly in the beginning and then it slows down. We have also added a vertical line at the point where all the seed values have been consumed and after which the fuzzer is only using generated values. All the different configurations are shown in the same graph to be able to compare them. We can observe that the Total Fuzzer, in both cases regardless of the driver, lags behind the Partial Fuzzer.

Contract	Par.Cov	Par.St	Par.Comb	Tot.Cov	Tot.St	Tot.Comb
AlgoTether	27.32	30.23	28.53	25.2	22.31	25.2
ExchangeToken	63.2	59.46	62.7	71.5	69.98	71.5
001	0.0	0.0	0.0	0.0	0.0	0.0
004	0.0	0.0	0.0	0.0	0.0	0.0
005	17.45	5.93	18.49	29.66	29.66	29.66
008	33.92	28.81	31.36	41.58	41.58	41.58
012	50.93	48.94	52.93	48.94	48.94	48.94
013	46.98	48.94	48.94	48.94	48.94	48.94
014	46.98	48.94	48.94	48.94	48.94	48.94
015	50.93	48.94	48.94	48.94	48.94	48.94
017	45.84	48.03	45.84	45.84	45.84	45.84
018	48.94	48.94	46.98	48.94	48.94	48.94
019	8.85	14.81	11.88	22.13	17.7	22.13
025	0.0	0.0	0.0	0.0	0.0	0.0

Table 4.1.: Final code coverage for each contract subtracting seed coverage.

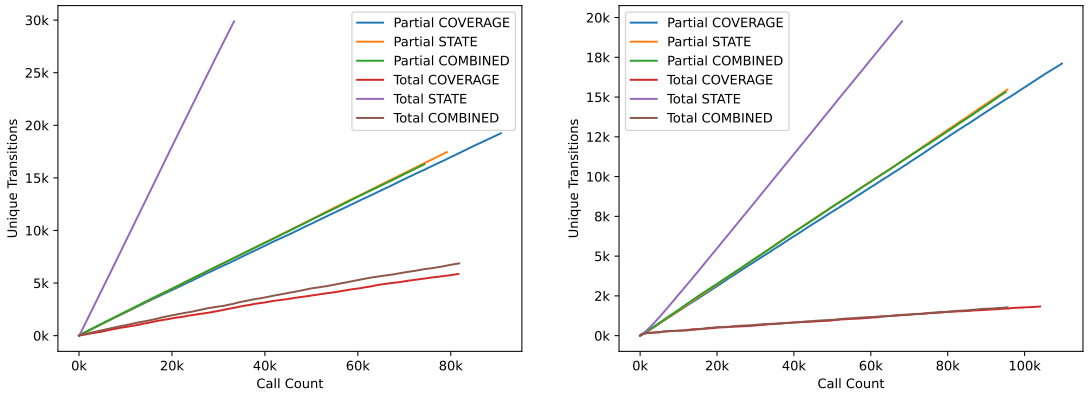
Another interesting observation is that the fuzzer behaves very differently for the two contracts in the early stages. For the AlgoTether contract, the fuzzer immediately covers new code on almost every call while consuming the seeds. On the other hand, for the ExchangeToken contract, most of the seed values do not cover new code. We also graphed the ranges and the medians of the coverage for each configuration separately,

which can be seen in Figure A.1 and Figure A.2 respectively. In these graphs can be seen that the coverage has no variance in the beginning for the Total fuzzer, while the Partial fuzzer has a lot of variance depending on the driver.

In Table 4.1 we can see the final code coverage for each contract after we remove the coverage gained by the seed inputs. Aside from contracts 1, 4, and 12, in all other contracts the fuzzers have gained coverage beyond what was covered by the seeds. Table A.1 shows us that almost all fuzzers achieved the maximum coverage. The only exception being the Total State fuzzer on the AlgoTether, ExchangeToken and '019' contracts, and Partial Coverage fuzzer on the AlgoTether contract.

State Discovery

For **RQ.2** we will look at the number of unique state transitions over the number of calls to the target contract. Although a new state transition does not necessarily mean that a new state was discovered, it is a good approximation (since the maximal number of transitions is equal to number of states squared). The best case scenario would be if the fuzzer discovers a new transition on every call. In Figure 4.2 we can see the unique transitions for the AlgoTether and ExchangeToken contracts. Here we confirm our previous observations that we made for the Total State fuzzer in the case study in section 3.6 which showed us that it is the most effective fuzzer at discovering new states. These results also show that using the Total State fuzzer introduces an overhead



(a) AlgoTether

(b) ExchangeToken

Figure 4.2.: Average transitions achieved by AlgoTether and ExchangeToken for all configurations.

since the number of calls is much lower even though the fuzzers run for the same amount of time. The other configurations of the Total fuzzer in this metric are heavily underperforming by comparison to the Total State fuzzer and the Partial fuzzer. They are barely reaching around 5000 or 2000 transitions, while the other fuzzers reach between 15000 and 30000 transitions. The Total fuzzer with the Coverage or Combined driver also introduces more variance in the number of transitions, as can be seen in Figure A.3 and Figure A.4.

To get a better understanding of the data we also calculated the ratio of transitions to calls for each configuration across all contracts. We calculated the ratio for each contract by dividing the average number of transitions by the average number of calls for each configuration on the last call. Best achievable ratio is 1.0 which would mean that the fuzzer discovered a new transition on every call. For this data we created box plots shown in Figure 4.3. The ratios are also shown for all contracts in Table A.3. In the box plot it is shown that the Total State fuzzer has the highest median ratio, around 0.9, making it near perfect in terms of discovering new transitions. The Partial fuzzers have a median of around 0.4 and the other Total fuzzers have a median of around 0.2.

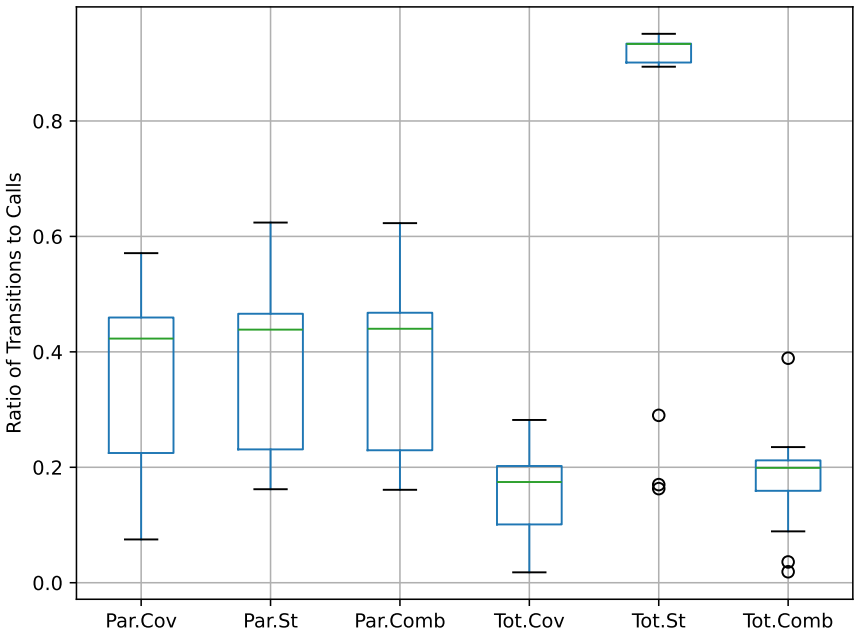


Figure 4.3.: Box plots for the ratio of transitions to calls for each fuzzer across all contracts.

5. Discussion

In this chapter we will discuss the findings of our study and the threats to the validity of the results.

5.1. Findings and Interpretations

RQ.1 Code Coverage

The results in section 4.2 show our approach was able to achieve a code coverage of 65% on average, which is comparable to results achieved in other studies for example Manticore achieved on average 66% code coverage on smart contracts [62]. The code coverage achieved is substantial also when we exclude the coverage achieved by the seed values. AlgoFuzz was able to achieve a code coverage of 32% on average when removing the code coverage achieved by the seeds. We have no other tool to compare this result to, but covering code beyond the seed values is a good indicator that AlgoFuzz is able to explore new code paths. Although 100% code coverage was not achieved in any of the runs, we assume this is due to the fact that some lines of code are not reachable, for example lines that are executed during the deployment of the smart contract.

Our data indicate the usual trend with regard to code coverage wherein the code coverage increases rapidly initially, but then it tends to plateau as the fuzzing session is extended. Because the fuzzer is designed to handle inputs that already conform to a specific structure (the official ABI), so it doesn't need to overcome the challenge of ensuring the inputs are correctly parsed by the program it's testing. Therefore, during the initial phases of fuzzing, the most straightforward and easily reachable code paths are explored. As most of the code paths are explored, the probability of finding new code paths diminishes. This of course is dependent on the structure of the smart contract being tested.

Answer to RQ.1: The results of our study demonstrate that AlgoFuzz is proficient in covering new code. In most instances, AlgoFuzz was able to attain the maximal coverage rate regardless of the configuration, meaning all configs achieved the same coverage in the end which we assume is the maximal coverage. Even when accounting for the coverage gained by the seed values, AlgoFuzz was able to achieve additional coverage. The best configuration gained on average an additional 34% coverage.

RQ.2 State Discovery

When dealing with state discovery, we cannot simply create a ratio of the number of unique states to the total number of states. This is because of the state explosion problem (or combinatorial explosion) which is a phenomenon where the number of possible states increases exponentially with the number of variables. Considering this, the total number of states would be in most cases much larger than the discovered states bringing the ratio close to zero.

To understand if AlgoFuzz is able to effectively discover new states, we analyzed the ratio of unique transitions to the number of calls made to the smart contract in section 4.2. Our data reveal that with the right configuration (the Total State configuration), AlgoFuzz is able to discover new transitions very effectively. As transitions and states are directly related, we can infer that AlgoFuzz is also able to discover new states effectively.

Answer to RQ.2: The results show that AlgoFuzz with the Total fuzzer and the State driven configuration is able to produce 0.9 unique transitions per call on average. Although, there is no other tool to compare this result to, considering that the maximum ratio is 1.0, we can conclude that AlgoFuzz is able to discover new states effectively.

5.2. Threats to Validity

In this section we will discuss the threats to the validity of our study. We differentiate between internal and external threats to validity. Internal validity referring to the correctness of the drawn conclusions and minimization of methodological errors. External validity refers to the extent the results can be generalized to other contexts.

Internal

As we already mentioned in 4.2, we had to exclude runs from the analysis. We assume that the LRZ server may have been under heavy load during the time of the runs. Other runs might have also been affected by the load. Although we currently have no way of knowing the impact of the load on the different runs, we cannot exclude that it had an impact on the results.

Another threat to internal validity is the non-deterministic behavior of the smart contract because of different blockchain states. We run the smart contracts against the same local Algorand network without resetting the network, since doing so after each run would have been too time-consuming. For this reason, the results of the runs may have been affected by not starting out with a clean state of the network.

External

The external validity of our study is limited by the number of smart contracts that we analyzed and by the fact that we only analyzed smart contracts which were ported to Algorand by us. AlgoFuzz cannot be used to analyze smart contracts which do not adhere to the official ABI. This is the case for the vast majority of the most popular smart contracts on Algorand [63]. Other limitations such as lack of dynamic storage support (box storage) and lack of support for ASAs also affect the external validity of our study.

The contracts we analyzed may also not be representative of the smart contracts that are currently deployed on Algorand. Contracts on Algorand may be more complex than the ones we analyzed. Aside from this, these contracts may have bugs which were introduced during the porting process.

In future ABI versions, more functionality may be added which may not be compatible with AlgoFuzz. This may limit the external validity of our findings for contracts which are written with an ABI version that is not currently supported.

6. Conclusion

In this thesis, we conceptualized, implemented and evaluated AlgoFuzz, a fuzzer for Algorand smart contracts. Smart contracts are one of the main building blocks of modern blockchains and are used to implement a wide range of dApps. They automate the execution of agreements between different parties and therefore control a large amount of financial value. For this reason, they are a prime target for attackers and their security is paramount.

Many software testing techniques have been proposed to detect vulnerabilities in smart contracts. One such technique is fuzzing, which has been shown to be effective in detecting vulnerabilities in traditional software. Tools that use fuzzing in regard to smart contracts have been proposed as well, and have shown great promise. However, most of these tools focus on the Ethereum blockchain, which is the most widely used blockchain for smart contracts.

In recent years, novel blockchain platforms that support smart contracts and address the shortcomings of Ethereum have emerged. The Algorand blockchain is one such platform, with features such as immediate transaction finality, high block creation speed and low transaction fees. Despite these advantages, there is still a lack of development tools for the Algorand ecosystem, making it difficult for developers to write secure smart contracts. Although static analyzers and testing frameworks have been introduced, a dedicated fuzzing tool for Algorand smart contracts has yet to be developed.

This thesis aims to address this gap by proposing AlgoFuzz. AlgoFuzz incorporates features like structured input generation, property-based testing, and multiple fuzzing strategies, enabling comprehensive testing of Algorand smart contracts. Through a series of empirical experiments, we evaluate the effectiveness of AlgoFuzz. Algorand smart contracts, derived from Ethereum smart contracts present in the Echidna benchmark suite, were used as primary testing grounds. Furthermore, two larger Algorand smart contracts were assessed. The results of the experiments show that on most configurations, AlgoFuzz is able to reach the highest achievable code coverage. Aside from some small contracts, AlgoFuzz was able to gain additional code coverage besides

the coverage gained by the seed values. The results also show that AlgoFuzz is able to discover new states effectively. With the right configuration for this metric, namely the Total State configuration, AlgoFuzz was able to discover on average 0.9 unique transitions per call.

Future Work

In this section we will discuss possible future work. We have divided this section into three subsections. Firstly we will discuss new features that could be added to AlgoFuzz. Secondly we will discuss how the limitations of AlgoFuzz could be addressed. Lastly we will go over how the evaluation of AlgoFuzz could be improved.

Features

Making seed values configurable would allow users to test specific values that they think may be problematic. This could be done in multiple ways either by letting the user specify the seed values for different data types or by specifying the seed values for methods for the contract. The user could potentially also specify the accounts that should be used for the runs, by choosing the account creating the contract, the accounts interacting with the contract, and the accounts that are passed as arguments to contract methods.

Another feature would be the ability to redeploy the contract during the fuzzing process. As it stands now, the contract is deployed once and then fuzzing is performed on the deployed contract. In the future, the contract could be redeployed to have different sequences of calls be executed on the contract. Potentially leading to different states of the contract and different code coverage. For such a feature, the user could for example specify the number of calls that should be executed before redeploying the contract.

AlgoFuzz could also be extended to support shrinking of failing inputs. This could be done by going backwards through the state transitions, starting from the failing state. This would allow the user to get a minimal transaction sequence for a failing run.

The fuzzer could also be enhanced to support static analysis methods which would allow the fuzzer to more efficiently explore different code paths.

Addressing Limitations

Currently, the `dryrun` endpoint is being phased out in favor of a new endpoint called `simulate`. This new endpoint does not yet support code coverage, but it is planned to be added in the future. The advantage here is that the `simulate` endpoint would also support box storage, which is one of the main limitations of AlgoFuzz.

Another limitation of AlgoFuzz is that it does not support ASAs. One way to address this is for the user to configure the fuzzer to use specific ASAs which are already deployed on the network. The use could pass a list of ASAs to the fuzzer which would then opt-in all accounts to the ASAs before starting the fuzzing process. Similarly, support for other reference type arguments could be added to the fuzzer.

When compiled with the Beaker framework, contracts not only include the usual ABI information but also provide hints about the different methods. These hints include what kind of transaction type each method supports, for example a method that should be called when the user is opting in to an application. With this information, the fuzzer could then support a wider range of contracts.

Evaluation

After alleviating some of the limitations of AlgoFuzz, the evaluation could be improved by running the fuzzer on a larger set of contracts. The goal of a future evaluation would be to test the fuzzer against contracts that are currently deployed on Algorand. As more and more contracts support the official ABI in the future, these contracts could be candidates for a new evaluation.

Another alternative to the current evaluation would be to find a set of contracts with know bugs. Then, carefully write property tests and fuzz the contracts to see if these properties would be violated.

A. Extensive Results

Contract	Par.Cov	Par.St	Par.Comb	Tot.Cov	Tot.St	Tot.Comb
AlgoTether	73.6	74.55	74.55	74.55	72.1	74.55
ExchangeToken	71.5	71.5	71.5	71.5	69.98	71.5
1	48.65	48.65	48.65	48.65	48.65	48.65
4	61.19	61.19	61.19	61.19	61.19	61.19
5	69.72	69.72	69.72	69.72	69.72	69.72
8	68.48	68.48	68.48	68.48	68.48	68.48
12	68.88	68.88	68.88	68.88	68.88	68.88
13	68.88	68.88	68.88	68.88	68.88	68.88
14	68.88	68.88	68.88	68.88	68.88	68.88
15	68.88	68.88	68.88	68.88	68.88	68.88
17	67.77	67.77	67.77	67.77	67.77	67.77
18	68.88	68.88	68.88	68.88	68.88	68.88
19	55.46	55.46	55.46	55.46	51.03	55.46
25	53.52	53.52	53.52	53.52	53.52	53.52

Table A.1.: Final code coverage for each contract.

Contract	Par.Cov	Par.St	Par.Comb	Tot.Cov	Tot.St	Tot.Comb
AlgoTether	19780.0	18601.33	18148.67	6895.33	32351.0	7439.83
ExchangeToken	17962.4	15696.0	15455.6	1941.6	20063.8	1833.2
1	4746.8	7598.4	7226.6	6379.5	6700.1	6661.6
4	22559.2	16142.1	15889.9	14463.2	17896.4	11252.0
5	15009.0	12532.4	12624.2	7257.4	16652.1	6586.8
8	13071.2	11800.0	11772.8	7463.4	16406.3	6855.5
12	15239.3	12529.3	12446.2	7939.0	17107.9	6580.9
13	14778.1	12198.6	12435.0	7290.3	16728.1	6127.8
14	14715.5	12473.7	12529.5	7766.2	16746.9	6514.7
15	14970.8	12575.9	12660.4	7734.1	16776.3	6556.9
17	14980.5	12175.3	12224.7	6297.2	16570.5	5613.1
18	14380.2	12333.4	12663.8	7673.3	16674.5	6765.5
19	9670.5	8652.4	9309.9	1483.9	16191.7	1482.2
25	4322.7	6920.6	7029.0	5448.1	6461.9	6195.4

Table A.2.: Average unique transitions for each configuration at the end of the runs.

Contract	Par.Cov	Par.St	Par.Comb	Tot.Cov	Tot.St	Tot.Comb
AlgoTether	0.212	0.221	0.219	0.073	0.894	0.089
ExchangeToken	0.156	0.162	0.161	0.018	0.290	0.019
1	0.080	0.212	0.198	0.104	0.170	0.165
4	0.571	0.624	0.623	0.282	0.951	0.389
5	0.414	0.459	0.457	0.172	0.935	0.210
8	0.415	0.428	0.426	0.207	0.923	0.235
12	0.458	0.467	0.469	0.204	0.934	0.212
13	0.444	0.448	0.451	0.177	0.934	0.188
14	0.460	0.466	0.464	0.196	0.935	0.212
15	0.461	0.466	0.470	0.197	0.934	0.211
17	0.431	0.429	0.429	0.154	0.933	0.163
18	0.465	0.473	0.470	0.204	0.934	0.219
19	0.263	0.261	0.261	0.033	0.924	0.036
25	0.075	0.199	0.190	0.100	0.163	0.158

Table A.3.: Ratio of transitions to calls at the end of the runs.

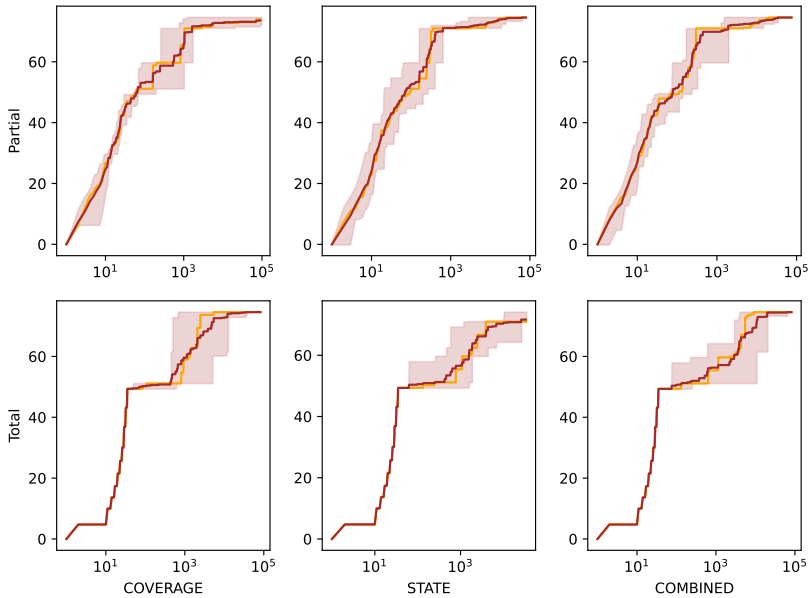


Figure A.1.: Code coverage for AlgoTether. (yellow - median, brown - mean)

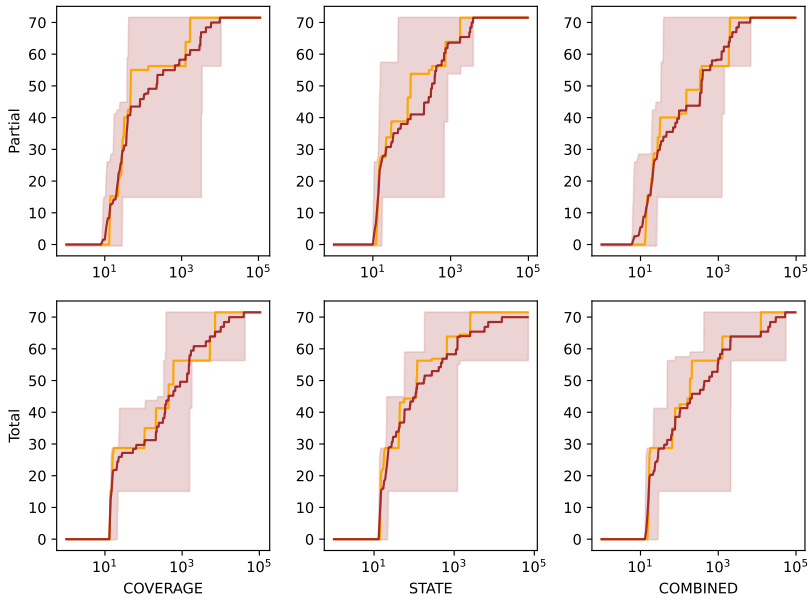


Figure A.2.: Code coverage for ExchangeToken. (yellow - median, brown - mean)

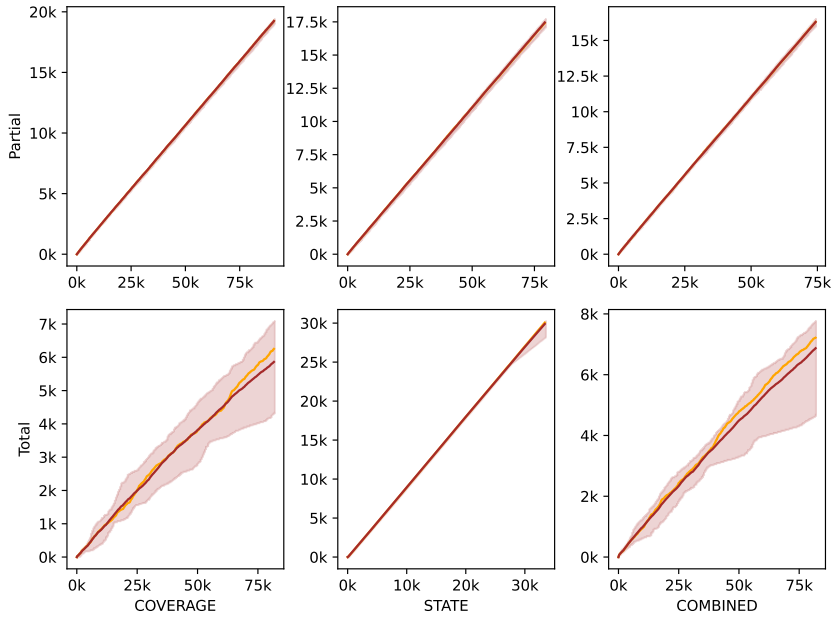


Figure A.3.: Unique transitions for AlgoTether. (yellow - median, brown - mean)

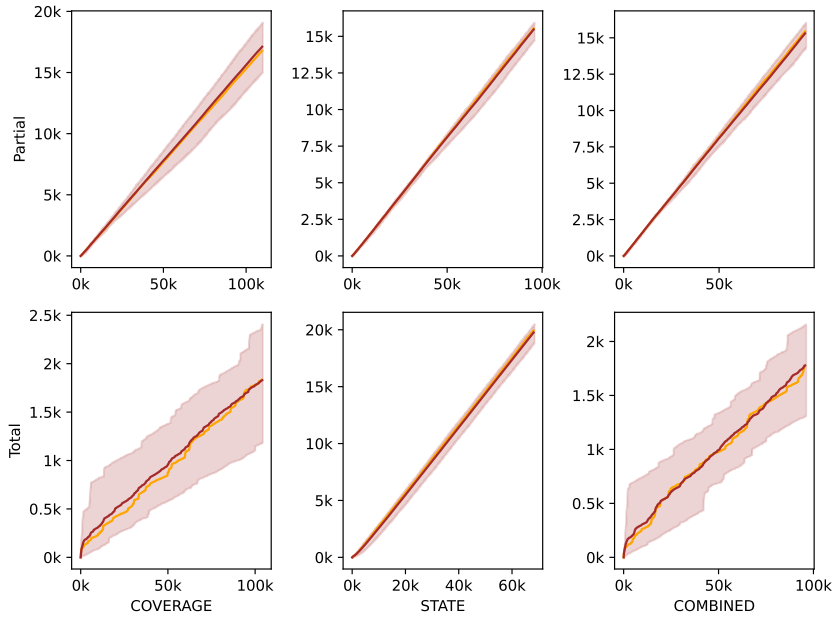


Figure A.4.: Unique transitions for ExchangeToken. (yellow - median, brown - mean)

Abbreviations

dApp Decentralized Application

PPoS Pure Proof of Stake

PoW Proof of Work

TPS Transactions per Second

MBR Minimum Balance Requirement

ASA Algorand Standard Asset

ABI Application Binary Interface

ARC Algorand Requests for Comment

TEAL Transaction Execution Approval Language

AVM Algorand Virtual Machine

CI Continuous Integration

CD Continuous Delivery

API Application Programming Interface

REST Representational State Transfer

SDK Software Development Kit

CSV Comma Separated Values

EVM Ethereum Virtual Machine

List of Figures

List of Tables

Bibliography

[1] *Global spending on blockchain solutions 2024*. Statista. URL: https://www.statista.com/statistics/800426/worldwide-blockchain-solutions-spending/ (visited on 08/20/2023).

[2] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. "Smart Contract Development: Challenges and Opportunities." In: *IEEE Transactions on Software Engineering* 47.10 (Oct. 2021). Conference Name: IEEE Transactions on Software Engineering, pp. 2084–2106. ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2942301.

[3] *Funds Lost Through Smart Contract Hacks in 2022 Stand at $2.7B, Representing a 1250% Jump Since 2020*. Bankless Times. URL: https://www.banklesstimes.com/news/2022/12/08/funds-lost-through-smart-contract-hacks-in-2022-stand-at-dollar27b-representing-a-1250percent-jump-since-2020/ (visited on 08/20/2023).

[4] C. Faife. *Wormhole cryptocurrency platform hacked for $325 million after error on GitHub*. The Verge. Feb. 3, 2022. URL: https://www.theverge.com/2022/2/3/22916111/wormhole-hack-github-error-325-million-theft-ethereum-solana (visited on 08/30/2023).

[5] *Smart contract security | ethereum.org*. URL: https://ethereum.org/en/developers/docs/smart-contracts/security/ (visited on 08/20/2023).

[6] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. *The Art, Science, and Engineering of Fuzzing: A Survey*. Apr. 7, 2019. DOI: 10.48550/arXiv.1812.00140. arXiv: 1812.00140[cs].

[7] T. Klooster, F. Turkmen, G. Broenink, R. t. Hove, and M. Böhme. *Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines*. June 7, 2022. DOI: 10.48550/arXiv.2205.14964. arXiv: 2205.14964[cs].

[8] J. Li, B. Zhao, and C. Zhang. "Fuzzing: a survey." In: *Cybersecurity* 1.1 (June 5, 2018), p. 6. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y.

[9] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. "Fuzzing: A Survey for Roadmap." In: *ACM Computing Surveys* 54.11 (Sept. 9, 2022), 230:1–230:36. ISSN: 0360-0300. DOI: 10.1145/3512345.

[10] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. "Echidna: effective, usable, and fast fuzzing for smart contracts." In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Event USA: ACM, July 18, 2020, pp. 557–560. ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3404366.

[11] *Echidna Trophies - The following security vulnerabilities were found by Echidna.* URL: https://github.com/crytic/echidna#trophies (visited on 08/20/2023).

[12] X. Guo. "Analysis between different types of smart contract fuzzing." In: *2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*. 2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA). May 2022, pp. 882–886. DOI: 10.1109/CVIDLICCEA56201.2022.9825021.

[13] J. Chen and S. Micali. "Algorand: A secure and efficient distributed ledger." In: *Theoretical Computer Science*. In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I 777 (July 19, 2019), pp. 155–183. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.02.001.

[14] *crytic/tealer: Static Analyzer for Teal*. GitHub. URL: https://github.com/crytic/tealer (visited on 08/30/2023).

[15] Z. Sun, X. Luo, and Y. Zhang. "Panda: Security Analysis of Algorand Smart Contracts." In: 32nd USENIX Security Symposium (USENIX Security 23). 2023, pp. 1811–1828. ISBN: 978-1-939133-37-3.

[16] *Algo Builder*. URL: https://algobuilder.dev/ (visited on 08/30/2023).

[17] P. S. Kochhar, F. Thung, and D. Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems." In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). ISSN: 1534-5351. Mar. 2015, pp. 560–564. DOI: 10.1109/SANER.2015.7081877.

[18] etherscan.io. *Tether USD (USDT) Smart Contract Code | Etherscan*. Ethereum (ETH) Blockchain Explorer. URL: https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d (visited on 09/02/2023).

[19] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." In: (2008).

[20] H. Brakmić. "Bitcoin Script." In: *Bitcoin and Lightning Network on Raspberry Pi: Running Nodes on Pi3, Pi4 and Pi Zero*. Ed. by H. Brakmić. Berkeley, CA: Apress, 2019, pp. 201–224. ISBN: 978-1-4842-5522-3. DOI: 10.1007/978-1-4842-5522-3_7.

[21] N. Szabo. "Smart Contracts: Building Blocks for Digital Markets." In: 1996.

[22] V. Buterin. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform." In: (2014).

[23] K. Wu. *An Empirical Study of Blockchain-based Decentralized Applications*. Feb. 13, 2019. DOI: 10.48550/arXiv.1902.04969. arXiv: 1902.04969[cs].

[24] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani. "Smart Contract Vulnerability Analysis and Security Audit." In: *IEEE Network* 34.5 (Sept. 2020). Conference Name: IEEE Network, pp. 276–282. ISSN: 1558-156X. DOI: 10.1109/MNET.001.1900656.

[25] pcaversaccio. *A chronological and (hopefully) complete list of reentrancy attacks to date.* GitHub. URL: https://github.com/pcaversaccio/reentrancy-attacks (visited on 08/30/2023).

[26] J. Chen and S. Micali. *Algorand*. May 26, 2017. DOI: 10.48550/arXiv.1607.01341. arXiv: 1607.01341[cs].

[27] *Algorand (ALGO) Blockchain Explorer*. URL: https://algoexplorer.io/ (visited on 08/30/2023).

[28] *Atomic transfers - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/atomic_transfers/ (visited on 08/30/2023).

[29] *Algorand Standard Assets (ASAs) - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/asa/ (visited on 08/30/2023).

[30] *Introduction - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/ (visited on 08/30/2023).

[31] *Reach*. URL: https://www.reach.sh/ (visited on 09/03/2023).

[32] *PyTeal: Algorand Smart Contracts in Python — PyTeal documentation*. URL: https://pyteal.readthedocs.io/en/stable/ (visited on 09/03/2023).

[33] *Beaker — Beaker documentation*. URL: https://algorand-devrel.github.io/beaker/html/index.html (visited on 09/03/2023).

[34] *@algorandfoundation/tealscript*. URL: https://tealscript.netlify.app/ (visited on 09/03/2023).

[35] *Contract storage - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/ (visited on 08/30/2023).

[36] *Overview - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/ (visited on 08/30/2023).

[37] *ARCs/ARCs/arc-0004.md at main · algorandfoundation/ARCs*. GitHub. URL: https://github.com/algorandfoundation/ARCs/blob/main/ARCs/arc-0004.md (visited on 08/30/2023).

[38] Crytic. *building-secure-contracts/not-so-smart-contracts/algorand at master · crytic/building-secure-contracts*. GitHub. URL: https://github.com/crytic/building-secure-contracts/tree/master/not-so-smart-contracts/algorand (visited on 08/23/2023).

[39] B. P. Miller, L. Fredriksen, and B. So. "An empirical study of the reliability of UNIX utilities." In: *Communications of the ACM* 33.12 (Dec. 1, 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[40] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. "Greybox Fuzzing." In: *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023.

[41] *Haskell Ethereum virtual machine evaluator*. URL: https://hevm.dev/ (visited on 09/03/2023).

[42] B. Jiang, Y. Liu, and W. K. Chan. "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Sept. 3, 2018, pp. 259–269. DOI: 10.1145/3238147.3238177. arXiv: 1807.03932[cs].

[43] V. Wüstholz and M. Christakis. "Harvey: a greybox fuzzer for smart contracts." In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 1398–1409. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3417064.

[44] ConsenSys AG. *Let's go next level smart contract security with Diligence Fuzzing.* Consensys Diligence. URL: https://consensys.io/diligence/fuzzing/ (visited on 09/03/2023).

[45] M. Ding, P. Li, S. Li, and H. Zhang. "HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection." In: *Evaluation and Assessment in Software Engineering*. EASE 2021. New York, NY, USA: Association for Computing Machinery, June 21, 2021, pp. 321–328. ISBN: 978-1-4503-9053-8. DOI: 10.1145/3463274.3463351.

[46] Y. Huang, B. Jiang, and W. K. Chan. "EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection." In: *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. Internetware '20. New York, NY, USA: Association for Computing Machinery, July 21, 2021, pp. 99–109. ISBN: 978-1-4503-8819-1. DOI: 10.1145/3457913.3457920.

[47] J. Zhou, T. Jiang, S. Song, and T. Chen. *AntFuzzer: A Grey-Box Fuzzing Framework for EOSIO Smart Contracts*. Nov. 2, 2022. DOI: 10.48550/arXiv.2211.02652. arXiv: 2211.02652[cs].

[48] *Home*. EOS Network. URL: https://eosnetwork.com/ (visited on 09/03/2023).

[49] *Hyperledger Fabric*. URL: https://www.hyperledger.org/projects/fabric (visited on 09/03/2023).

[50] Algorand. *algorand/tealfuzz*. GitHub. URL: https://github.com/algorand/tealfuzz (visited on 09/03/2023).

[51] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy. "Checking Smart Contracts with Structural Code Embedding." In: *IEEE Transactions on Software Engineering* 47.12 (Dec. 1, 2021), pp. 2874–2891. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2020.2971482. arXiv: 2001.07125[cs].

[52] *AlgoKit - Algorand Developer Portal*. URL: https://developer.algorand.org/docs/get-details/algokit/ (visited on 09/05/2023).

[53] *hone-labs/teal-interpreter: An interpreter for TEAL assembly code that simulates the Algorand virtual machine.* GitHub. URL: https://github.com/hone-labs/teal-interpreter (visited on 09/06/2023).

[54] *@algo-builder/runtime*. URL: https://algobuilder.dev/api/runtime/index.html (visited on 09/06/2023).

[55] *Algorand Python SDK*. URL: https://developer.algorand.org/docs/sdks/python/ (visited on 09/18/2023).

[56] Michał Zalewski. *Binary fuzzing strategies: what works, what doesn't*. Aug. 8, 2014. URL: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html (visited on 05/21/2023).

[57] *Algorand (ALGO) On-Chain Analytics & Charts*. URL: https://app.intotheblock.com/coin/ALGO/deep-dive?group=financials&chart=avgTrxSize (visited on 07/15/2023).

[58] M. Böhme, V.-T. Pham, and A. Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428.

[59] *ERC-20: Token Standard*. Ethereum Improvement Proposals. URL: https://eips.ethereum.org/EIPS/eip-20 (visited on 09/18/2023).

[60] *kupl/VeriSmart-benchmarks*. GitHub. URL: https://github.com/kupl/VeriSmart-benchmarks (visited on 09/21/2023).

[61] *LRZ Compute Cloud*. Leibniz-Rechenzentrum (LRZ). Section: Dokumentation. URL: https://doku.lrz.de/compute-cloud-10333232.html (visited on 09/21/2023).

[62] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). ISSN: 2643-1572. Nov. 2019, pp. 1186–1189. DOI: 10.1109/ASE.2019.00133.

[63] *Algorand Applications*. URL: https://algoexplorer.io/applications (visited on 10/04/2023).