NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Static Analysis for Detecting Side Effects in Ethereum Smart Contracts

Vasileios E. Poulopoulos

Supervisor: **Yannis Smaragdakis,** Professor NKUA

ATHENS

JULY 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Στατική Ανάλυση για Ανίχνευση Παρενεργειών σε Έξυπνα Συμβόλαια Ethereum

Βασίλειος Η. Πουλόπουλος

Επιβλέπων: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2022

BSc THESIS

Static Analysis for Detecting Side Effects in Ethereum Smart Contracts

Vasileios E. Poulopoulos

S.N.: 1115201600141

SUPERVISOR: **Yannis Smaragdakis,** Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Στατική Ανάλυση για Ανίχνευση Παρενεργειών σε Έξυπνα Συμβόλαια Ethereum

Βασίλειος Η. Πουλόπουλος
Α.Μ.: 1115201600141

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ABSTRACT

Ethereum is the most widespread blockchain in which smart contracts are executed. In recent years, the applications that are built using smart contracts have gained mass appeal that keeps growing.

Gigahorse is a framework that decompiles EVM bytecode and can run static analyses on smart contracts.

The goal of this thesis is to produce useful statistics about the presence of side effects in the public functions of smart contracts in the main net of Ethereum, in order to document the consistency of the programming of public functions, acknowledge suspicious practices and have a global picture when auditing smart contracts with public signatures.

SUBJECT AREA: Static Analysis

KEYWORDS: ethereum, static analysis, side effects, smart contracts, public signatures

ΠΕΡΙΛΗΨΗ

Το Ethereum είναι το πιο διαδεδομένο blockchain στο οποίο εκτελούνται smart contracts. Τα τελευταία χρόνια, η δημοτικότητα των εφαρμογών που δημιουργούνται με τη χρήση των smart contracts έχει αυξηθεί κατακόρυφα.

Το Gigahorse είναι ένα Framework, μέσω του οποίου μπορεί να γίνει decompilation από EVM bytecodes αλλά και στατικές αναλύσεις για smart contracts.

Σκοπός της πτυχιακής εργασίας είναι να παραχθούν στατιστικά για το κατα πόσο υπάρχουν side effects σε public signatures, με σκοπό να καταγραφεί η συνέπεια στον προγραμματισμό των public functions, να αναγνωριστούν ύποπτες συμπεριφορές τους και να υπάρχει μία σφαιρική εικόνα όταν ελέγχονται smart contracts με γνωστά signatures.

Για να γίνει αυτό υλοποιήθηκε μία ανάλυση σε souffle η οποία βρίσκει τα side effects στα public signatures από κάθε smart contract στο mainnet του Ethereum και στη συνέχεια υπολογίστηκαν τα στατιστικά, σε τι ποσοστό δηλαδή εμφάνισης ενός public signature υπάρχουν side effects.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ethereum, στατική ανάλυση, παρενέργειες, έξυπνα συμβόλαια, δημόσιες υπογραφές

To my family.

ACKNOWLEDGEMENTS

I would like to thank my Professor Yannis Smaragdakis, Iosif Lagouvardos and Ilias Tsatiris for their assisntance and patience throughout the preparation of this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis was developed as my Bachelor thesis at the National and Kapodistrian University of Athens at the department of Informatics and Telecommunications. The idea behind the subject was to collect useful metrics about public signatures that help analyze the code when performing security audits. The technologies that were used were Soufflé, Python, and Gigahorse.

1. INTRODUCTION

Since the creation of Bitcoin [6], cryptocurrencies and blockchain technology have been gaining more and more popularity. Every day new ones are being created and used. Ethereum [4] extended the use of the blockchain technology by enabling its users to run code in its virtual machine. These pieces of code are called smart contracts. Smart contracts deal with billions of dollars, hence smart contract security is very important.

This thesis's goal is to provide useful statistics about the existences of side effects in commonly used public functions in order to identify possible vulnerabilities and predict these functions' behavior in future audits.

To implement this, we write static analyses in datalog to identify side-effects and run it through the Gigahorse [5] decompiler for the smart contracts that are deployed in the one million most recent blocks of ethereum. Then, we calculate statistics for every public signature.

Let's give a brief overview of the rest of the thesis:

- In Chapter 2 we provide background on cryptography, blockchain, cryptocurrencies, Ethereum and smart contracts, solidity, static analyses and the Gigahorse decompiler.
- In Chapter 3 we provide the methodology that we followed to write the analysis and the scripts that calculate the statistics
- In Chapter 4 we describe the procedures we followed to evaluate the results
- In Chapter 5 we demonstrate the results
- In Chapter 6 we provide a summary of the thesis and conclusions.

2. BACKGROUND

2.1 Cryptography

Blockchains rely a lot on cryptography, hence before discussing them we should address some things about cryptography and more specifically about hash functions.

2.1.1 Hash Functions

A hash function is a function, that accepts a numerical input of any length and outputs a compressed numerical output of a fixed length.

2.1.1.1 Hash Function Properties

Hash functions have three properties:

- They have **collision resistance**, meaning that two input values cannot have the same output.
- They have **pre-image resistance**, meaning that given the output, it should be nearly impossible to compute the input.
- They have **second pre-image resistance**, meaning that given an input and its hash, it should be very difficult to find another input that leads to the same hash.

Some very well-known and widely used hashing functions are MD5, SHA256, SHA1, SHA2, SHA3. They are used for various purposes, one of which is storing passwords in a database. Instead of the passwords, their hashes are stored, and when a user inputs his password, the input is hashed, and the two hashes are compared.

In Ethereum hashing is used for various reasons and the keccak256 hashing algorithm is preferred.

2.1.2 Public-key cryptography

Public-key cryptography is a cryptographic system that utilizes pairs of keys that consist of a public and a private key. The public key is publicly known but the private key is only known by the owner.

This kind of cryptography is used to send messages in a network. The private key is used to sign the message and the public key to validate it. In Ethereum, for example, when an individual wants to send ether, they use the receivers' public key as the address that the ether will be sent to, and signs the transaction using his private key.

2.2 Blockchain

A blockchain is a distributed database, that occurs in a peer to peer network instead of a centralized server. The data that are collected are stored in groups known as blocks.

Blocks have a certain capacity, and every block contains a hash that links it to the previous block. The first block of a blockchain is called the genesis block and instead of a hash of a previous block, it has a zero. This architecture creates a database that is essentially a linked list of blocks, hence the name.

2.3 Cryptocurrencies

Cryptocurrencies are virtual currencies that are used to transact goods. They do not rely on trusted third parties to process the payments, thus they are theoretically immune to manipulation. Most cryptocurrencies rely on the blockchain technology to store their transactions. Each block contains a set of transactions.

The first cryptocurrency was Bitcoin that was introduced to the world in 2008, by an unknown developer who goes by the name of Shatoshi Nakamoto creating a big wave of interest on the blockchain technology. Since then new cryptocurrencies are being created every day and are gaining more popularity and usability.

2.4 Ethereum and Smart Contracts

Ethereum [1], extends the possibilities of the blockchain technology, not only by storing transactions of its native cryptocurrency ”Ether”, but also executing programs called smart contracts. Ethereum consists of a globally accessible singleton state and a virtual machine, the Ethereum Virtual Machine, that applies changes to it.

Smart contracts are deployed in the ethereum blockchain by sending a transaction to 0x0 which is reserved for that purpose. Every execution of a smart contract is considered a transaction in the blockchain. One key feature of smart contracts, is that they are immutable, meaning that their code cannot change after deployment.

Ethereum uses blockchain to store the system's state changes, along with the transactions of ether. Ether is essentially a utility currency to pay for the use of the Ethereum platform.

Smart contracts cannot instantiate a transaction like an EOA can. They can instead send messages, that are essentially a special type of transactions if they want to send a transaction to another account.

Ether has subdivision's, with the smallest one being the wei, where $1ETH = 10^{18}wei$.

In Ethereum there are individual accounts called Externally Owned Accounts (EOAs). Smart Contracts as well as Externally Owned Accounts, can hold ether. Both have unique addresses in order to interact and be identified through the network. Externally Owned Accounts, are controlled by their private keys, in contrast to smart contracts that are controlled by their code.

Just like Bitcoin, Ethereum has a revolutionary essence because of the introduction of smart contracts. Countless blockchains have emerged since then that can run smart contracts, but Ethereum is the most commonly used.

The two most used and maintained languages for smart contract development are Solidity and Vyper.

2.5 Solidity

The most used programming language for smart contract development on Ethereum, is without a doubt Solidity[2]. It is a high level object-oriented with its grammar being very similar to popular languages like C++, Python and Javascript. It provides inheritance, advanced user defined types and a lot more features one can find in most languages.

2.5.1 Solidity Program Structure

Let's demonstrate the basic structure of solidity programs that is needed for the purpose of this thesis, through the following example.

It is essentially a simplified version of an ERC-20 token, a token that lives in the Ethereum blockchain and follows the EIP-20 prototype.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract ERC20 {
    string public constant name = "ERC20";
    string public constant symbol = "ERC";
    uint8 public constant decimals = 18;

    mapping(address => uint256) balances;

    event Transfer(address indexed from, address indexed to, uint256 tokens);

    uint256 public immutable totalSupply;

    constructor(uint256 total) {
        totalSupply = total;
        balances[msg.sender] = total;
    }

    function balanceOf(address tokenOwner) public view returns (uint256) {
        return balances[tokenOwner];
    }

    function transfer(address receiver, uint256 numTokens) public returns
        (bool) {
        require(balances[msg.sender] >= numTokens);
        balances[msg.sender] = balances[msg.sender] - numTokens;
        balances[receiver] = balances[receiver] + numTokens;
        emit Transfer(msg.sender, receiver, numTokens);
        return true;
    }

    function transferFrom(address owner, address buyer, uint256 numTokens)
        public returns (bool) {
        require(balances[owner] >= numTokens);
        balances[owner] = balances[owner] - numTokens;
        balances[buyer] = balances[buyer] + numTokens;
        emit Transfer(owner, buyer, numTokens);
        return true;
    }
}
```

Figure 2.1: Solidity Example

As seen above, every contract has a name and wraps every element of it. There are

- events
- variables
- functions
- data structures
- modifiers

Modifiers are special properties of functions that are used to define the behavior of the function. They can be defined by the programmer of the smart contract, but there are some predefined ones too. For example, a predefined modifier, is `payable`. When a function is defined as payable, it means that the smart contract can receive ether through this function. An example of a user defined modifier would be `onlyOwner`. Before the

```
modifier onlyOwner {
  require(msg.sender == owner);
  _;
}
```

Figure 2.2: onlyOwner modifier

function runs, the modifier is run. If the requirement succeeds, the function runs.

Every public function or variable of a smart contract must be declared with a visibility modifier:

- `external` - External functions can be called via transactions from other smart contracts or EOAs.
- `internal` - Internal functions can only be called by the contract itself or contracts that inherit from it.
- `public` - Public functions can either be called internally or externally.
- `private` - Private functions are like the internal ones, but are not visible to the contract that inherit from it.

Variables must be declared with the same identifiers except the `external` one.

The `msg` variable is a special global variable that exists in every solidity smart contract and contains useful information, like the address that calls a function, using `msg.sender`, or the amount of wei , that is used for the transaction, using `msg.value`. In this thesis we are interested in public functions, and more precisely those with a known public signature, explained below.

Except for the normal functions there is the constructor that initializes the contract and an empty function, named the fallback function that is run when the function that is called does not exist.

2.6 Ethereum Virtual Machine

In order for the code to be executed in different machines Ethereum utilizes the Ethereum Virtual Machine that is the core of the Ethereum network. Smart contracts, need to be compiled in EVM bytecode. When a smart contract is deployed, its bytecode is stored on the blockchain and is connected to its address.

2.6.1 Storing data

There are two ways to store data in a smart contract. One can either use memory, or storage. Storage is used for variables that need to be permanently stored, and is part of Ethereum's state.

In the example above, the fields of the contract need to be permanently stored, so they are stored in storage.

```
        string public constant name = "ERC20";
        string public constant symbol = "ERC";
        uint8 public constant decimals = 18;
        mapping(address => uint256) balances;
        uint256 public immutable totalSupply;
```

Figure 2.3: Solidity Storage

The low lever instruction that is being used when data has to be stored in storage is SSTORE and to load data from storage is SLOAD.

Memory is for storing temporary data that gets lost when the execution of a contract ends. To store data in the memory the low level instruction is MSTORE and and to load them is MLOAD.

2.6.2 Application Binary Interface

Application Binary Interface (ABI), is an interface between two binary program modules.

As mentioned above, a smart contract in EVM is just a sequence of bytecode. A relation between the bytecode and the high level languages is needed in order to access the functions of a contract. Moreover, a conversion of the return value is needed in order to be understandable by users.

In Ethereum, ABI is used to connect the bytecode with the higher level languages, in order to make the method calls reliable.

Every public method's name and arguments are hashed using Keccak-256. This hash is called the function signature, and the first four bytes are used to identify the function.

2.6.3 Calls

Solidity offers low level functions for calling other contracts that have the same name with the EVM opcodes that correlate directly to. These functions are call and delegatecall, that replaced the deprecated callcode.

Calling another contract can cause a large number of security related vulnerabilities. For example, using call can lead to reentrancy.

Delegatecall is mostly used to call functions from libraries. It is even more dangerous than call because it keeps the same msg.sender as in the calling contract, whereas call changes the msg.sender to be the calling contract. In simpler words, delegatecall runs the code of the contract that was called inside the context of the caller contract, which can potentially lead a malicious user to take advantage of this functionality in order to change another contract's state.

Callcode was even riskier because it would not preserve the msg.sender and msg.value.

2.6.4 Self destruct

As mentioned before a smart contract's code cannot be changed. Yet, if a smart contract is programmed that way, the code can be deleted from the ethereum's state, leaving a blank account in its address. Smart contract deletion can be done by executing low level EVM opcode called SELFDESTRUCT.

2.7 Static Analysis

Static analysis is, in contrast to dynamic analysis that is performed when a program is executed, a program analysis that is performed without executing the program. Although dynamic analyses can be more precise, they cannot scale in large programs with a lot of possibilities, and that is why static analysis is useful even though it is not so accurate.

Datalog, a subset of prolog, is a commonly used declarative logic programming language for static analyses. A datalog rule can be stated in any order without altering the results. Furthermore, performance plays a big role in why datalog is preferred for such actions.

2.7.1 Gigahorse-toolchain

Gigahorse-toolchain is a framework that is built as a platform to run security analyses on EVM smart contracts.

Gigahorse is a reverse compiler of Ethereum Virtual Machine smart contracts' code to a higher level three address representation (3-address code intermediate representation).

On top of that, the gigahorse-toolchain runs analyses on the three address code representation as well as user generated analyses too. The analyses' results are extracted in csv files in a format that makes it easy to pass them as inputs in other programs, written in any language, for further analytics and statistics

Gigahorse-toolchain utilizes datalog and more specifically the Soufflé[3] implementation, that is a fast datalog engine that compiles datalog programs into C++ ones.

Compared to its competition, Gigahorse is more scalable, gets better results, and is more time efficient, with a significantly smaller amount of contracts timing out.

The process which Gigahorse follows, splits the bytecode into basic blocks, and then, a local analysis is performed to find stack effects per block.

3. METHODOLOGY

In programming, a function is considered to have a side effect when it modifies anything outside of its parameters and local variables in order to function correctly.

When auditing Ethereum code, it would be very useful to know the probability of a function having side effects only based on its public signature in order to identify its behavior and possible vulnerabilities.

As side-effects on Ethereum smart contract we consider the following:

- CALLS
- SSTORE
- SELFDESTRUCT

In order to calculate the statistics the following steps had to be done:

1. Get the unique bytecodes from every smart contract that has been deployed in the Ethereum blockchain.
2. Run a static analysis, that identifies public functions that have side effects and return their public signatures.
3. Calculate statistics and save the results for every public signature if further inspection is needed.

3.1 Static Analysis for Side Effects Discovery

For the static analysis part of this thesis, the following datalog code was used in order to collect the public functions with side effects signatures:

```
1   .decl FunctionHasSideEffects(fun: Function)
2   FunctionHasSideEffects(fun) :-
3     (CALL(stmt, _, _, _, _, _, _, _, _);
4     DELEGATECALL(stmt, _, _, _, _, _, _, _);
5     CALLCODE(stmt, _, _, _, _, _, _, _);
6     SSTORE(stmt, _, _);
7     SELFDESTRUCT(stmt, _)),
8     Statement_Block(stmt, block),
9     InFunction(block, fun).
10
11  FunctionHasSideEffects(fun) :-
12    FunctionHasSideEffects(other),
13    CallGraphEdge(callerBlock, other),
14    InFunction(callerBlock, fun).
15
16  .decl PublicFunctionSigHasSideEffects(fun: Function, sigText: symbol)
17  PublicFunctionSigHasSideEffects(fun, sigText) :-
18    FunctionHasSideEffects(fun),
19    PublicFunctionSelector(fun, pubSig),
20    ConstantPossibleSigHash(pubSig, sigText).
```

Figure 3.1: Datalog Rules for Side Effects Discovery

Let's examine the above datalog code. First of all, the `FunctionHasSideEffects` rule is created to find every function that has a side effect.

The first query of `FunctionHasSideEffects`:

1. Checks for statements of calls, stores in storage and for self destructions as stated above. (lines 3-7)
2. Finds the basic block in which the found statement exists. (line 8)
3. Finds the function in which the basic block exists and returns it. (line 9)

The second query of `FunctionHasSideEffects`:

1. Finds a function (`other`) that contains a side effect. (line 12)
2. Checks for transitivity. More accurately, it finds the basic blocks that flow to that function. (line 12)
3. Finds the function in which each basic block of the above exists and returns it. (line 14)

The last query of the analysis, `PublicFunctionSigHasSideEffects`,filters the results and excludes functions that are not public with known signatures, leaving us only with useful information about public signatures.

Let’s examine `PublicFunctionSigHasSideEffects` further too:

1. Finds functions with side effects based on `PublicFunctionSigHasSideEffects`. (line 18)
2. Finds its signature (the first four bytes of the keccak256 hash as described in the previous chapter). (line 19)
3. Finds its known name. (line 20)

3.2 Results Extraction

For this step a script was created in order to collect the results and display them in a better form.

The script implements the following:

1. Outputs a csv file that contains the signature names, the number of times that signatures are found and the percentage of the side effects.
2. Creates a directory called contracts-with-side-effects. Inside, there is a csv with the name of every public signature. Each of them, contains the bytecode md5 hashes of the contracts that have side effects in that particular signature.
3. Creates a directory called contracts-without-side-effects with the same structure as the contracts-with-side-effects directory, that contains a file for every signature with all the bytecode md5 hashes of the contracts that have this signature and do not have side effects.
4. Lastly, another directory that contains files for every signature and every file contains the bytecode md5 hash of every contract that contains said signature.

4. EVALUATION

The evaluation of the results is a bit ambiguous because it cannot be fully automated and needs a lot of manual inspection. False results could arise for two reasons:

- The analysis is imprecise and misses cases that side effects exist
- The decompiler produces false results, so the analysis that runs on top of it, even though it works as expected, misses the cases that the decompilation is wrong.

The approaches that were followed to find the accuracy of the analysis were two.

- Manually inspect smart contract that have source code and check if any unexpected results are to be found.
- Collect a random sample of 100 public functions and check if they are defined with one of the `view` and `pure` modifiers. In that case, our analysis should have them flagged as side effect free and if not, as side effect positive. The view/pure annotation is our source of truth. Thus, we assume that view/pure functions are side effect free and the rest are not. Then calculate precision and recall scores.

4.1 Manually inspect contracts

For this method, the approach was to find public signatures that should be 100% side effect free or 100% not side effect free but were not and inspect manually the suspicious contracts.

This way, false positives can be found too and we can take a closer look to what is operated.

The negative aspect of this method is that it cannot be automated and it is dependent on human inspection. It is obvious that it cannot be done for all or even most of the public signatures.

4.2 Sample view/pure functions for scores calculation

Following the manual inspection method, the idea now is to randomly collect a sample of 100 public signatures and check if they are defined with a `view` or `pure` modifier in their source code. We know that these functions are 100% side effect free. The functions that do not have these annotations, should be side effect positives. Then we check if these public functions are flagged by our analysis as side effect free or not, and split the sample into true positives, true negatives, false positives and false negatives.

This method if very useful, as it can give us metrics about how accurate our analysis is. In the following chapter we demonstrate the `precision` and `recall` results that the sample gave us.

4.3 Metrics

In the next chapter, the metrics are presented in the following manner:

- The statistics of the analysis
- Results based on the manual inspection
- Score results based on sampling

5. RESULTS

In this chapter, we examine the results of the process that we discussed in the previous chapters. For dataset, we used all the bytecodes of the smart contracts of the last one million blocks of the Ethereum blockchain in the time of writing this thesis. The total number of the dataset's bytecodes are 79138.

5.1 Getting the results

For demonstration purposes, the 30 most frequent used signatures are presented below:

Table 5.1: Most frequent signatures

Name	Side Effects Percentage	Frequency
owner()	0.02%	32825
name()	0%	30289
symbol()	0%	29583
balanceOf(address)	0.22%	29259
transferOwnership(address)	99.96%	29222
transferFrom(address,address,uint256)	98.37%	28825
approve(address,uint256)	99.21%	28787
renounceOwnership()	99.08%	28711
totalSupply()	0.06%	26933
supportsInterface(bytes4)	0.03%	19091
isApprovedForAll(address,address)	0.02%	17136
setApprovalForAll(address,bool)	99.67%	17091
tokenURI(uint256)	0.33%	15996
ownerOf(uint256)	0.02%	15850
safeTransferFrom(address,address,uint256)	99.74%	15712
safeTransferFrom(address,address,uint256,bytes)	99.73%	15709
getApproved(uint256)	0.01%	15702
decimals()	0.03%	13276
transfer(address,uint256)	96.78%	13241
allowance(address,address)	0.13%	13068
withdraw()	99.91%	10142
setBaseURI(string)	99.96%	9771
tokenOfOwnerByIndex(address,uint256)	0%	8923
tokenByIndex(uint256)	0%	8867
increaseAllowance(address,uint256)	99.45%	8485
decreaseAllowance(address,uint256)	99.46%	8478
paused()	0.05%	8038
mint(uint256)	99.88%	6713
maxSupply()	0%	5550
burn(uint256)	99.5%	4385

5.2 Evaluating the results

Let us consider the `owner()` and check the validity of the results.

Our analysis considers that the following contracts `owner()` functions have side effects:

Table 5.2: Owner contracts with side effects

Bytecode md5 hash
88E9B569AC40CE09029C84FBFD302387
52FA28BF372D59E3A6FFCBF2DBFB6C41
63AA34EC2528C8EF27A2B8F44B28D2D3
9676EB0B8B7D6D58BC91960BE8EFD2A9
D0D36E832E5E011F5AE22CA8EB814859
E5FAC20011CDF19EB5024140CDF566C5
3B696008A6DECE627FFA8A3FD95D7BA7
5D2AAC5F6DD191D5CA21E243920B0D24

Let us check the contracts that are left manually. By looking at the following contracts source codes, we can easily identify that they are falsely flagged as side-effect positives.

Table 5.3: view/pure false positives

Bytecode md5 hash
52FA28BF372D59E3A6FFCBF2DBFB6C41
63AA34EC2528C8EF27A2B8F44B28D2D3
9676EB0B8B7D6D58BC91960BE8EFD2A9
D0D36E832E5E011F5AE22CA8EB814859

Let us take the first contract of the above for example. The `owner()` function in the source code of this contract is the following:

```
function owner() public view virtual returns (address) {
  return _owner;
}
```

Figure 5.1: owner() source code

We can clearly identify that it is a side effect free function.

Now we check the decompiled representation of the same function:

```
1   function owner() public nonPayable { find similar
2       if (msg.sender == _owner) {
3           require(_wETH.code.size);
4           v0, v1 = _wETH.balanceOf(this).gas(msg.gas);
5           require(v0); // checks call status, propagates error data on error
6           require(MEM[64] + RETURNDATASIZE() - MEM[64] >= 32);
7           if (v1) {
8               require(_wETH.code.size);
9               v2, v3 = _wETH.transfer(msg.sender, v1).gas(msg.gas);
10              require(v2); // checks call status, propagates error data on error
11              require(MEM[64] + RETURNDATASIZE() - MEM[64] >= 32);
12              require(v3 == v3);
13              if (v3) {
14                  emit
                        0xbbdff98e8d8bd134663fc8d9254d063f8dda646873259d52807579c571f9b348(1,
                        v1);
15              }
16          }
17          if (this.balance) {
18              0x182e(this.balance, msg.sender);
19              emit
                    0xbbdff98e8d8bd134663fc8d9254d063f8dda646873259d52807579c571f9b348(0,
                    this.balance);
20          }
21          exit;
22      }
23      revert('Ownable: caller is not the owner', 'Ownable: caller is not the
            owner');
24  }
```

Figure 5.2: owner() decompiled code

In line 9 a transfer function is called. The decompilation is clearly not accurate in this case, and as a result our analysis identifies the `owner()` as side-effect positive.

The other four contracts are considered as false positives because of a bug in the decompilation too.

We identified half of the flagged contracts as false positives and we are left with the following contracts:

Table 5.4: view/pure unknown

Bytecode md5 hash
88E9B569AC40CE09029C84FBFD302387
E5FAC20011CDF19EB5024140CDF566C5
3B696008A6DECE627FFA8A3FD95D7BA7
5D2AAC5F6DD191D5CA21E243920B0D24

Checking manually the above four contracts to inspect if those are false positives too is not possible because their solidity code is not published.

Most likely, these four too, are false positives, because of the nature of the `owner()` function. It should not have any side effects.

Nevertheless, the accuracy of our analysis seems to be very high, especially in the case of `owner()`, because out of the 32825 only 8 of them were flagged as side effect positives.

All the public signatures that are displayed here, are very common and it is easy to guess if they should have side effects or not. Functions like `owner()`, `name()` and `symbol()`, just return a value that they read from the state, so one would expect that they are 100% side effect free, and our analysis shows results that approximate to 0% side effects, indeed. Functions like `transferOwnership(address)` and `transferFrom(address,address,uint256)` on the other hand, at least use SSTORE to store the changes into the Ethereum state, so they cannot be side-effect free. Our analysis results agree with that too, showing such function having 99% side effects.

5.3 Evaluation Scores

For the collection of a sample of 100 public signatures, we focused on the signatures that are more frequent. After collecting them randomly, only two of them were false positives and one false negative.

The sample contains:

- 50 true positives (Functions that are view/pure and were identified by the analysis as side-effect free)
- 2 false positives (Functions that are not view/pure but were identified by the analysis as side-effect free)
- 47 true negatives (Functions that are not view/pure and were identified by the analysis as side-effect positives)
- 1 false negatives (Functions that are view/pure but were identified by the analysis as side-effect positives)

Below the precision and recall scores are presented:

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} = \frac{50}{52} = 0.96$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} = \frac{50}{51} = 0.98$$

When we inspected the 2 false positives and 1 false negative manually, we realized that they were falsely flagged due to the bug in the decompilation that we came across in the previous section and not due to some buggy rule in our analysis. Inspecting further false positives and false negatives, we couldn't find a case that was wrong due to our analysis. On the contrary, all of them were caused by bad decompilation.

6. CONCLUSIONS

Smart contracts are a new technology and is getting used more and more by the day. We presented side-effects statistics of the most common public signatures and we found out that our analysis is very accurate hitting precision and recall scores as high as 96% and 98% respectively.

To check the validity of our analysis, we picked a sample of public functions of contracts that had their source codes available. Then with the `view` or `pure` modifiers being our source of truth, we knew that if they were defined as view/pure they were side effect free and otherwise they were not, so we calculated precision and recall scores.

ABBREVIATIONS - ACRONYMS

EVM	Ethereum Virtual Machine
EOA	Externally Owned Account
ABI	Application Binary Interface
ERC	Ethereum request for comment
EIP	Ethereum Improvement Proposals

APPENDIX A. FIRST APPENDIX

1	FunctionHasSideEffects
2	PublicFunctionSigHasSideEffects

BIBLIOGRAPHY

[1] Mastering ethereum. `https://github.com/ethereumbook/ethereumbook`.

[2] Solidity documentation. `https://docs.soliditylang.org/en/v0.8.13/`.

[3] Soufflé. `https://souffle-lang.github.io/`.

[4] Vitalik Buterin. Ethereum whitepaper. 2013. `https://ethereum.org/en/whitepaper/`.

[5] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. `https://bitcoin.org/bitcoin.pdf`.