NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Machine learning aided tuning of static analysis for EVM bytecode decompilation

Markella A. Gioka

Supervisors: **Yannis Smaragdakis,** Professor NKUA
Iosif Lagouvardos, M.Sc. Student NKUA
Ilias Tsatiris, M.Sc. Student NKUA

ATHENS

SEPTEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Προσαρμογή στατικής ανάλυσης για αποσύνθεση κώδικα EVM με χρήση μηχανικής μάθησης

Μαρκέλλα Α. Γκιόκα

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Ιωσήφ Λαγουβάρδος, Μεταπτυχιακός Φοιτητής ΕΚΠΑ
Ηλίας Τσατίρης, Μεταπτυχιακός Φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΗΣ 2020

BSc THESIS

Machine learning aided tuning of static analysis for EVM bytecode decompilation

Markella A. Gioka

S.N.: 1115201400269

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
Iosif Lagouvardos, M.Sc. Student NKUA
Ilias Tsatiris, M.Sc. Student NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Προσαρμογή στατικής ανάλυσης για αποσύνθεση κώδικα EVM με χρήση μηχανικής μάθησης

Μαρκέλλα Α. Γκιόκα

Α.Μ.: 1115201400269

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Ιωσήφ Λαγουβάρδος, Μεταπτυχιακός Φοιτητής ΕΚΠΑ
Ηλίας Τσατίρης, Μεταπτυχιακός Φοιτητής ΕΚΠΑ

ABSTRACT

The Gigahorse toolchain, utilizing the declarative approach to static program analysis, offers a reverse compiler of smart contracts of the Ethereum blockchain. The selection of call-site context depth, a problem commonly encountered in static analyses, has proven challenging in the setting of Ethereum Virtual Machine bytecode, in part due to its low level design which is optimised for execution on the blockchain.

The aim of this thesis is to provide a good estimation of the context depth parameter which minimizes the analysis time while also maintaining a high accuracy of the decompilation. To achieve this, we used a quick pre-analysis to extract features from smart contracts and chose a machine learning model which has a good performance in estimating the optimal context depth per contract.

SUBJECT AREA: Static program analysis

KEYWORDS: context-sensitive analysis,
program analysis, machine learning, Ethereum

ΠΕΡΙΛΗΨΗ

Η αλυσίδα εργαλειών του Gigahorse, εφαρμόζοντας την δηλωτική προσέγγιση για στατική ανάλυση, προσφέρει έναν αντίστροφο μεταγλωττιστή για smart contracts. Η επιλογή του ύψους των call-site συμφραζομένων, ένα πρόβλημα που εντοπίζεται συχνά στον τομέα της στατικής ανάλυσης, έχει αποδειχθεί πρόκληση για τον κώδικα της εικονικής μηχανής του Ethereum, εν μέρη λόγω του χαμηλού επιπέδου σχεδιασμού του, ο οποίος είναι βελτιστοποιημένος για εκτέλεση στο Ethereum blockchain.

Σκοπός της παρούσας πτυχιακής εργασίας είναι να παράξουμε μία ακριβής εκτίμηση της παραμέτρου του ύψους των συμφραζομένων η οποία να ελαχιστοποιεί τον χρόνο της ανάλυσης διατηρώντας ταυτόχρονα υψηλή ακρίβεια για την αποσύνθεση. Για να το επιτύχουμε αυτό, κάνουμε χρήση μιας γρήγορης προ-ανάλυσης με σκοπό την εξαγωγή χαρακτηριστικών των smart contract και εν συνεχεία κάνουμε χρήση ενός μοντέλου μηχανικής μάθησης το οποίο επιδεικνύει καλή απόδοση στην πρόβλεψη του βέλτιστου ύψους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση με συμφραζόμενα,
ανάλυση προγραμμάτων, μηχανική μάθηση, Ethereum

To my family.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for giving me the opportunity to do research on such an interesting topic, and for all of the support throughout the development of this thesis.

I am also very thankful to Iosif Lagouvardos and Ilias Tsatiris for their patience and assistance which have been instrumental in completing this work. Lastly, I would like to acknowledge NKUA research fellow Neville Grech for his useful insights and recommendations on this project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Gigahorse [6] is at its core a decompiler for the Ethereum [12] Virtual Machine (EVM) bytecode, employing the declarative approach to static analysis. Context-sensitive static analysis qualifies abstract objects such as stack states with context information, for example call-sites. For instance, in k-call-site context sensitive analysis, objects are qualified by k call sites. A context-sensitive analysis allows for separate computations for executions that account to different contexts. This has the advantage of added precision when different contexts result in different program behaviour. However, context-sensitive analysis can become unscalable when there is an unpredictable explosion of contexts, leading to prohibitive levels of complexity. Determining the ideal context depth per program is an optimization problem of achieving good precision and maintaining scalability.

In the setting of call-site context sensitivity, the number of call sites that can be statically detected in the code has been used in the past by algorithms for tuning the context depth. In the case of EVM bytecode, context depth tuning is hindered even more by its low-level stack-based design, as jump targets are variable values read from the stack. The focus of our work is to answer the question of whether a machine learning approach can be used to effectively estimate the optimal depth for each given contract. To achieve this, we define critical features that can be easily extracted by a quick pre-analysis as well as a simple algorithm that determines the optimal context depth of a contract. Using those two, we can create a training dataset and use it to evaluate the effectiveness of different machine learning models.

This thesis is organized as follows:

1. In Chapter 2 we provide some background on the Ethereum platform, the EVM bytecode, the Gigahorse toolchain and context sensitivity in static analysis
2. In Chapter 3 we describe the methodology and meaning of the features used, and the algorithm for determinig the optimal depth
3. In Chapter 4 we present and evaluate the machine learning models considered
4. In Chapter 5 we showcase our experimental results of decompilations under different contexts
5. In Chapter 6 we summarize our conclusions

2. BACKGROUND

2.1 The Ethereum Platform

Ethereum [12], also described as a “world computer”, is an open source decentralized blockchain with the key property of being programmable. Its two main components are a globally accessible singleton state and a virtual machine which supports code execution. Through transactions and code execution, changes can be applied to the global state. Although Ethereum is not unique in supporting a scripting language, unlike other blockchains, for instance Bitcoin, its virtual machine allows the execution of code of unbounded complexity, namely Turing complete. The programs that can be executed on Ethereum’s Virtual Machine (EVM) called smart contracts, can be developed in a number of high-level languages, such as Solidity or Vyper, and then be compiled to EVM bytecode. The compiled code can then be deployed on the platform with a transaction sent to the address reserved for contract creation, which is 0x0.

Contracts, along with Externally Owned Accounts (EOA) are a kind of account on Ethereum. Both types of accounts have a unique Ethereum address. The main difference between the two is that EOAs are controlled by users while contract accounts are controlled by the code associated with them. However, contracts can only be executed by being called by a transaction, usually by having their address set as the recipient of the transaction, which was initiated by an EOA. In turn, a contract can call another contract and trigger its execution, but the initial call must have been made from an EOA.

To counter risks arising from EVM’s bytecode Turing completeness like non halting execution causing the abuse of resources, Ethereum incorporates a charging mechanism called gas. Each instruction has a predetermined amount of gas that will be consumed by executing it, and each transaction has a set amount of gas that has been paid upfront. This sets an upper limit to the amount of resources an execution processes can consume per transaction, since its execution will forcibly terminate when the available gas is spent. Gas fees can be paid in Ethereum’s digital currency, Ether.

Furthermore, a contract’s code cannot be modified once it is deployed. Even though a contract can self destruct by executing a special instruction, that would still only remove the code segment, leaving behind an empty account and the account’s transaction history.

2.2 EVM Bytecode

At the core of Ethereum is its virtual machine, which is an important factor of what sets it apart from other decentralized systems. Compared to other virtual machines, it serves a similar purpose and that is to provide compatibility across multiple different underlying systems. With execution happening on a public resource, though, there are additional considerations for efficiency. This has led to the introduction of abstractions in the EVM not commonly found in other Intermediate Representations (IR), thus adding some difficulty to the task of reverse compilation. In this section we will take a look at EVM’s bytecode general design and distinctive properties and demonstrate some of those properties with examples.

2.2.1 Overview

Solidity, the dominant choice by developers for the writing of smart contracts [3], is a high-level, contract-oriented language with syntax similar to JavaScript and C++. As such, it is enhanced with many high-level features like function calls, types and contract inheritance. In the listing below we see a simplified version of an auction app in solidity [2].

```
contract AuctionDapp {
    address payable public beneficiary;
    address ownerAddr;
    address public highestBidder;
    uint public highestBid;
    bool ended;
    constructor(
        address payable _beneficiary
    ) {
        beneficiary = _beneficiary;
        ownerAddr = msg.sender;
    }
    function bid() public payable {
        require(msg.value > highestBid);
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    function auctionEnd() public  {
        require(msg.sender == ownerAddr);
        require(!ended);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

Figure 2.1: Example contract in Solidity

The EVM, in contrast, has a very low level design, featuring a stack-based architecture with no notion of types or structures, only a 256 bit (or 32 byte) word. Its instruction set includes operations in the following categories:

- Arithmetic
- Comparison and bitwise logic
- SHA3
- Accessing environment information
- Accessing block information
- Memory, storage and flow
- Stack manipulation
- Logging
- System operations

Besides the stack, contracts also have a Memory and a Storage for data storage and memory operations. Storage is persistent memory and is considered part of the Ethereum state. For example, for the snippet above, all of the fields will be saved in the contract's Storage.

```
contract AuctionDapp {
    address payable public beneficiary;
    address ownerAddr;
    address public highestBidder;
    uint public highestBid;
    bool ended;
(...)
}
```

Figure 2.2: Contract persistent storage

Memory is freshly instantiated for each message call and is used for temporary storage. Storage can be accessed with the EVM instructions `SLOAD/SSTORE`, while `MLOAD/MSTORE` are used for accessing the memory, respectively.

EVM does not have the concept of labels, instead, all possible jump destinations are specified by the `JUMPDEST` instruction. Hence, basic blocks begin after a jump or a jump destination.

Instructions have to use the stack to read their arguments and to return any results. A simple example is an arithmetic operation that takes two arguments. It will pop the top two contents of the stack, execute the operation, and then push the result back on the stack. The following listing adds two integers and compares the result with a third integer, demonstrating the stack states for each step. It should be noted that EVM's stack follows the conventional last-in, first-out (LIFO) protocol.

```
    PUSH1 0x1
    PUSH1 0x2
    ADD
    PUSH1 0x3
    EQ
```

Figure 2.3: Stack based operations example

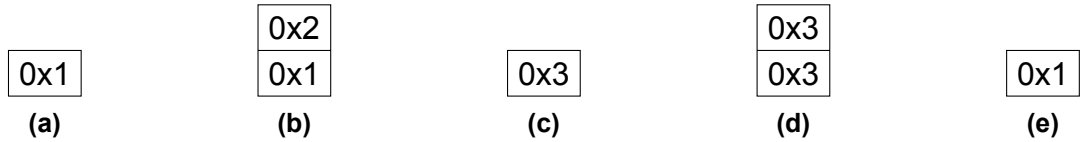

Figure 2.4: Stack states

This convention includes jump instructions, which means that jump destinations are not constant but, rather, variable values read from the stack. The destination address can not always be determined by simply examining the latest value pushed on the stack, as in some cases the address gets pushed way before the jump instruction, and is retrieved with stack manipulation instructions such as `DUP` and `SWAP`. As a consequence, control flow edges are dependent on the stack contents under different executions, meaning an

accurate control flow graph cannot be constructed without performing context-sensitive value flow analysis.

Public functions are uniquely identified by the first four bytes of their signature's hash, which is a Keccak-256 hash of the function's name and arguments. This is part of the Application Binary Interface (ABI), which allows public functions to be called externally by providing the function's hash as an argument in the call. For our example, the function hashes would be computed as such:

```
{
    "2a24f46c": "auctionEnd()",
    "38af3eed": "beneficiary()",
    "1998aeef": "bid()",
    "d57bde79": "highestBid()",
    "91f90157": "highestBidder()"
}
```

Figure 2.5: Public function signatures

The reason the fields `beneficiary`, `highestBid` and `highestBidder` have corresponding functions is because they have been declared as public, which assigns them with a public function that returns their saved value.

Lastly, at the beginning of the EVM runtime bytecode exists a code segment not explicitly programmed in the original code, called the function selector. By disassembling our example using the disassembler provided by Gigahorse, we get the following:

```
    (...)
    0xf: CALLDATALOAD
   0x10: PUSH29    0x100000000000000000000000000000000000000000000000000000000
   0x2e: SWAP1
   0x2f: DIV
   0x30: DUP1
   0x31: PUSH4     0x1998aeef
   0x36: EQ
   0x37: PUSH2     0x6c
   0x3a: JUMPI
   0x3b: DUP1
   0x3c: PUSH4     0x2a24f46c
   0x41: EQ
   0x42: PUSH2     0x76
   0x45: JUMPI
   0x46: DUP1
   0x47: PUSH4     0x38af3eed
   0x4c: EQ
   0x4d: PUSH2     0x8d
   0x50: JUMPI
   0x51: DUP1
   0x52: PUSH4     0x91f90157
   0x57: EQ
   0x58: PUSH2     0xce
   0x5b: JUMPI
   0x5c: DUP1
   0x5d: PUSH4     0xd57bde79
   0x62: EQ
   0x63: PUSH2     0x10f
   0x66: JUMPI
   (...)
```

Figure 2.6: Function selector

The above segment executes once every time a contract is called. It loads the signature that was passed with the contract's call, and then it repeatedly compares that with the public functions' signatures until it finds a match, in which case it executes the appropriate jump. We can observe the hashes that correspond to our functions being pushed on the stack.

As it is evident from the previous listing, EVM has no special instructions for performing intra-contract function invocation and returns. High level function calls get implemented as jumps, with function arguments and the return addresses being placed on the stack. Thus, separating function calls from simple jumps and denoting function boundaries in EVM bytecode is far from trivial and requires significant program understanding in order to deduce from code behaviour.

2.3 The Gigahorse Toolchain

The Gigahorse Toolchain provides a reverse compiler for the transformation of EVM bytecode to a functional 3-address code representation. This representation incorporates some of the structural language characteristics which are naturally lacking from EVM bytecode. For example, variables are introduced and private function calls are specified formally with call and return arguments. Therefore, the carrying out of a variety of client

analyses is enabled, which can be implemented in any language by using as input the decompilation results as CSV files.

Compared to previous EVM decompilers, Gigahorse exhibits better precision and scalability, yielding results in substantially better times for a higher percentage of the existing contracts. That performance is highly attributed to the declarative static-analysis based approach.

2.3.1 Declarative Static Analysis

The term static analysis exists to create a distinction from dynamic analysis. Usually, dynamic analysis evaluates a program by examining its behaviour under different inputs, while static analysis is performed by examining only the source code. Although dynamic analyses can be more precise for programs they can analyse, for complex programs they tend to become both unscalable and lose precision since it gets harder to test for all possibilities.

Declarative analysis is the term used when a declarative language, for instance Datalog as is the case for Gigahorse, is used for the implementation. Declarative languages express the logic of a program as a set of rules, in contrast to imperative languages which express the flow of execution. A rule is made up of two parts, the head of the rule and the body. If the body consists of a set of facts that have been evaluated as true, then the head is also true, i.e. it gets inferred. The inference of rules, in turn, can lead to more inferences until all possible statements get evaluated, which is called the least fixpoint.

Datalog is favoured in program analyses tasks for a couple of reasons. The logic rules have no set order of execution, and any order will provide the same results. This property allows the examination and understanding of different parts of the program in isolation, as its context should not have an effect on its execution. In addition, probably more importantly, Datalog simplifies the specification of recursive definitions, which is an important property for program analysis tasks as they usually consist of mutually recursive sub-tasks.

Datalog is also preferred for performance reasons. The independent nature in which the rules are evaluated makes them ideal for mass parallelization and optimization. Most importantly, though, such concerns do not greatly affect the programmer, since execution is passed, for the most part, to the language interpreter or compiler. Gigahorse, in particular, uses the Soufflé [7] implementation, a Datalog engine designed specifically for program analyses, which translates rules to C++ programs and uses a variety of optimization techniques.

2.3.2 Gigahorse Logic Overview

The input files for the analysis contain bytecode, as it is generated by a smart contract language compiler and is executed on the blockchain. That bytecode contains the opcode values of the instructions as well as static values in the code in binary form. Extracting from the binary file instruction names and values is not a challenging task, as there is a one to one mapping of opcodes to EVM instructions. Before the main logic of the decompilation can begin, a procedure that is called fact generation has to be applied. Fact generation takes the bytecode and produces information in a format that can be used by the analysis program. Specifically, it produces the following input relations.

```
S is a set of statement identifiers
V is a set of variables intruduced by Gigahorse
B is a set of basic block identifiers
C is a set of constants
```

```
Statement_Opcode(s:S, opcode:C)
Statement_Next(prev:S, next:S)
PushValue(s:S, v:C)
PublicFunctionSignature(hex: C, text: C)
EventSignature(hex:C, text:C)
```

Figure 2.7: Gigahorse input relations

We will give a brief overview of the high-level structure of Gigahorse's rules. The first step in the bytecode transformation is to identify blocks in the code and perform context-insensitive analysis locally on the block level, including examination of the stack effects per block. This step is quite straightforward and not too computationally expensive, as complexity is influenced linearly only by the number of lines of code in the contract and there is no unpredictable recursiveness that can affect its scalability. In the next step, context-sensitive analysis is performed on the whole program, with control-flow graph construction and two Intermediate Representations (IR) are introduced. The value-flow analysis of this step is directly dependent on the context, as objects like block stack input states are qualified by their context information. The first IR produced is referred to as *global 3-address IR* and represents operations with the use of global registers. With the help of this IR, function boundaries are inferred heuristically, enabling the construction of local control-flow graphs as well as the production of the final *functional 3-address IR*.

2.4 Context Sensitivity

Context sensitivity is one of the most standard ways to add precision to static analysis [10]. Call-site context-sensitive analysis, which is the kind that Gigahorse utilizes, qualifies abstract objects and variables, such as stack contents, with a sequence of k call-sites. In the context of EVM bytecode, the notion of calls is replaced by jumps and call sites by blocks. We shall examine a simple example to illustrate how call-site context can enhance precision.

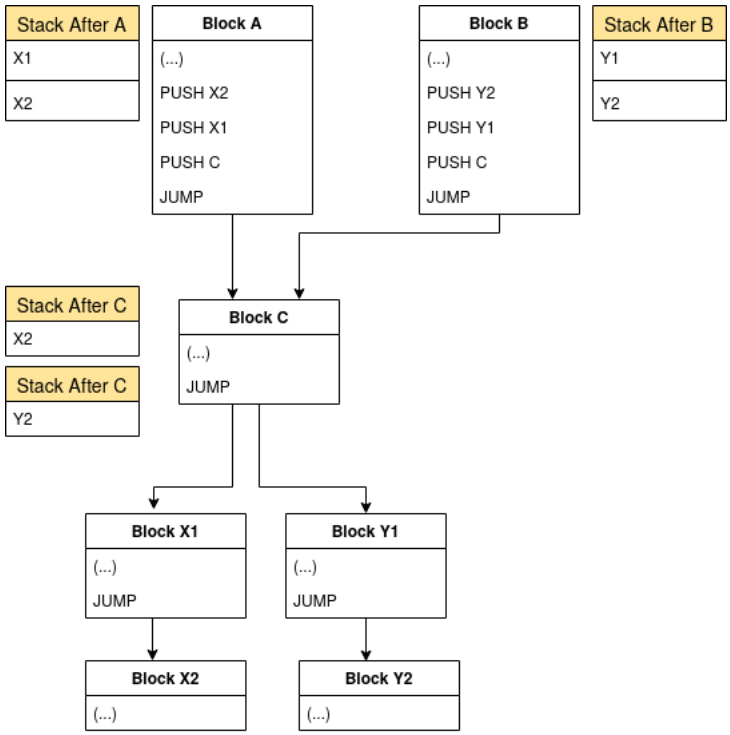

Figure 2.8: Control flow example

Initially, blocks A and B push two addresses on the stack and then both jump to block C. Next, block C will perform a jump to the address on the top of the stack, which would be either X1 or Y1. Blocks X1 and Y1 will also use the stack contents to perform a jump. Of course, this is a simplified scenario, but we can imagine C being a sequence of blocks, instead, that have a total delta on the stack that is equal to zero.

We can see that block C has two possible stack state outputs, depending on which block precedes it. If we use 1-call-site context, X1 and Y1 will both have context [C], and therefore their stack inputs will be all of the possible stack outputs of C. As a result, both blocks will have edges to both X2 and Y2.

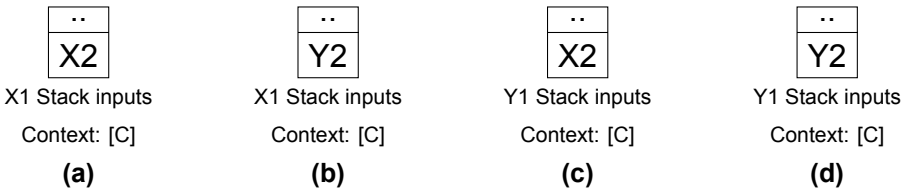

Figure 2.9: 1-call-site context stack states

If we analyse this with a 2-call-site context, the contexts [A,C] and [B,C] account to a different stack state each. In this case, X1 is only reachable from [A,C], and Y1 from [B,C], respectively, thus each block only having one possible jump target.

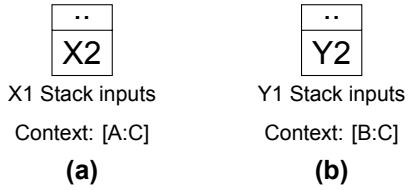

Figure 2.10: 2-call-site context stack states

The jumps on X1 and Y1 in the 1-call-site context example are called polymorphic since we can not resolve them to one jump target. Function return jumps are typically excluded from that definition as they always depend on the caller. Potentially, C could be considered to have a polymorphic jump as well, although that is a pattern that we would commonly encounter in function returns.

One thing the previous example fails to highlight is the source of unpredictable unscalability associated with context sensitivity. What we witnessed in the example is the best case scenario in which the added context perfectly separates the computations according to the context information. However, there could be a case in which the extra context does not alter the program behaviour. The result of such case would be the computations getting multiplied by the number of possible contexts, and thus increasing the time and space complexity.

3. PREPROCESSING

3.1 Feature Extraction

In the previous chapter we gave an overview of the Ethereum platform and the Gigahorse toolchain in order to explain how call-site context affects both the scalability and precision aspects of decompilation. Next, we will present the features considered for the tasks of training and prediction, while also examining the logic rules that define the pre-analysis used for the extraction of those features. The main purpose of the pre-analysis is to provide us with quick-to-acquire, high-level features that satisfiably reflect the relative complexity of each contract.

3.1.1 Basic Blocks

We will begin by first looking at features that describe the basic structure of a smart contract. As mentioned in the previous section about the EVM architecture, in EVM bytecode a block begins:

- At a `JUMPDEST` instruction
- After a `JUMP` instruction

Both of the above things can be statically identified in the bytecode. This way we can get as a high level feature the count of blockheads in the contract.

3.1.2 Basic Block Stack Effects

Having split the bytecode in basic blocks it becomes possible to examine the effects of each block on the stack.

3.1.2.1 Stack Delta

As discussed earlier, instructions take their arguments from the stack and return their results also on the stack. That follows that all instructions can both push a certain amount of items and pop a certain amount from the stack, both of which are set. The difference between the amount pushed and the amount popped for each instruction is considered to be the instruction's *stack delta*, and is indicative of whether the stack contents were increased or decreased. For example, the instruction `ADD` would have a delta of -1, since it pops 2 items and pushes 1, resulting in an item less in the stack in total. In this way, by summing all the stack deltas of each instruction per block we can get the block's delta.

3.1.2.2 Stack Pop Delta

This feature follows a similar logic to the simple stack delta but serves a different purpose. Stack pop delta tries to answer the question *how many items from the caller's stack will this block consume?* The key idea is to keep a maximum of the number of elements

consumed from the stack that were not pushed from any of the block's instructions. To explain simply how this computation takes place we'll consider an example.

```
1    JUMPDEST
2    ADD
3    ADD
4    (...)
```

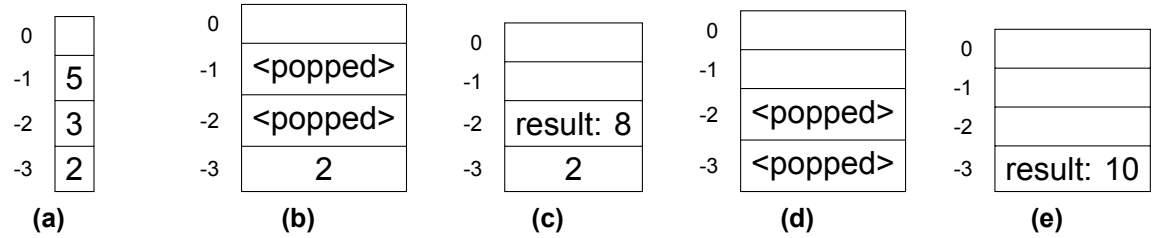

Figure 3.1: Example for pop delta

As we can see, after the first `ADD` instruction, two items have been consumed and one item was returned on the stack. The stack was empty before the instruction, so both of the items consumed were taken from the calling block's stack. All in all, though, the stack is lowered by one position. As a result, we have a pop delta of 2 and a stack delta of -1. By the second `ADD` instruction, two items are consumed, again, only this time, one of them was a value that was locally produced by the block. Therefore, only one item was consumed from the caller's stack. The result was also placed on the stack. Finally, that accounts to a pop delta of 3 and a stack delta of -2. This feature is an indicator of how heavily a block depends on previous blocks' products, possibly reflecting the need for call-site context for achieving ideal precision in value flow analysis.

```
BlockPopDelta(block: B, delta: number)
```

```
MaxPopDelta(n:number)
MaxPopDelta(n):-
 n = max delta : BlockPopDelta(_, delta).

HighPopDelta(n:number)
HighPopDelta(n):-
 IsPopStackIndexInRange(delta,5,10),
 n = count : BlockPopDelta(_,delta).

VeryHighPopDelta(n:number)
VeryHighPopDelta(n):-
 IsPopStackIndexInRange(delta,10,16),
 n = count : BlockPopDelta(_,delta).

C_HighPopDelta(n:number)
C_HighPopDelta(n):-
  n = sum d : HighPopDelta(d).

C_VeryHighPopDelta(n:number)
C_VeryHighPopDelta(n):-
  n = sum d : VeryHighPopDelta(d).
```

Figure 3.2: Pop delta predicates

3.1.3 Gas Related

Each EVM instruction costs a set amount of gas in order to execute. Not all instructions cost the same amount of gas, with some having magnitudes larger requirements than others. The basic idea behind the gas prices is to price the instructions according to the computational resources they consume. This type of evaluation could give us a high-level description of a contract's complexity, as the more complex the instructions it includes are, the higher the gas total will be. One problem, though, is that some instructions we would not consider complex in terms of program logic, are priced high because they might be using the storage or memory which are also scarce resources and priced very highly. In the table bellow we give some examples of gas consumption.

Table 3.1: Example gas prices

Name	Description	Gas
`ADD`	Addition	3
`MUL`	Multiplication	5
`SHA3`	Keccak-256 hash	30
`JUMP`	Alter program counter	8
`JUMPI`	Alter program counter conditionally	10
`SLOAD`	Load word from storage	200
`CREATE`	Create new account	32000

After noticing that most instructions we are interested in have a cost of 40 or lower, we define a different computation for gas. Instead of adding all the gas values of each instruction as is, we simply assign a cut-off limit of 50, meaning that any instruction cost higher than that gets accounted for a cost of 50.

```
Statement_Opcode(stmt: S, opcode: C)          //Opcode of statment s
OpcodeGas(opcode: C, gas: number)             //Gas of opcode op
```

```
Statement_Gas(stmt:S, gas:number)
Statement_Gas(stmt, gas):-
    Statement_Opcode(stmt, opcode),
    OpcodeGas(opcode, gas),
    gas < 50.
Statement_Gas(stmt, 50):-
    Statement_Opcode(stmt, opcode),
    OpcodeGas(opcode, gas),
    gas >= 50.

Block_PartialGas(block: B, stmt: S, gas: number)
Block_PartialGas(block, stmt, gas) :-
    Statement_Block(stmt, block),
    Statement_Gas(stmt, gas).

Block_Gas(block: B, gas: number)
Block_Gas(block, totalgas) :-
    Statement_Block(_, block),
    totalgas = sum gas : Block_PartialGas(block, _, gas).

MaxBlockGas(gas: number)
MaxBlockGas(gas):-
  gas = max block_gas: Block_Gas(_,block_gas).

TotalGas(gas: number)
TotalGas(gas):-
  gas = sum block_gas: Block_Gas(_,block_gas).
```

Figure 3.3: Gas predicates

3.1.4 Basic Block Connectivity

Next, we will gather some information regarding the connectivity of the contract's blocks. However, resolving all the jumps is a task that requires extensive and context-sensitive value flow analysis, so we will have to use only the edges that are easily decided on the local stack analysis level.

```
JUMPDEST(s: S)
Statement_Next(s: S, n: S)
IsBsicBlockHead(s: S)          \\ Statement s is a block head
Statement_Block(s: S, b: B)    \\ Statement s belongs to block B
ImmediateBlockJumpTarget(b: B, v: V, s: S)
FallThroughStatement(s: S)     \\ Statement s is fallthough
Variable_Value(var: V, val: C) \\variable v has value val
```

```
BlockJumpValidTarget(block:B, var: V, target:B)
BlockJumpValidTarget(block, targetVar, target) :-
  ImmediateBlockJumpTarget(block, targetVar),
  Variable_Value(targetVar, target),
  IsBasicBlockHead(stmt),
  Statement_Block(stmt, target),
  JUMPDEST(stmt).

BlockEdge(caller: B, callee: B)
BlockEdge(caller, callee):-
    BlockJumpValidTarget(caller,_,callee).

BlockEdge(caller, callee):-
    FallThroughStatement(stmt),
    Statement_Block(stmt,caller),
    Statement_Next(stmt, fallthrough),
    IsBasicBlockHead(fallthrough).

ReachableBlock(from: B, to: B)
ReachableBlock(from, to) :-
    BlockEdge(from, to).

ReachableBlock(from, to) :-
    ReachableBlock(next, to),
    BlockEdge(from, next).

ReachableFrom(to: B, n: number)
ReachableFrom(to, n):-
    Statement_Block(_, to), n = count : ReachableBlock(_, to).

MaxFrom(n:number)
MaxFrom(n):-
    n = max n1 : ReachableFrom(_, n1).
```

Figure 3.4: Reachability predicates

The predicates at the top of the above listing are borrowed from Gigahorse's local analysis, so we will not get into them in much detail. Gigahorse represents stack indices as variables in order to perform in later stages value flow analysis. In this way, it is able to differentiate which variables are local to the block and which are not. `ImmediateJumpTarget` denotes the jumps that can be resolved to be on a local variable. A fallthrough statement is just a statement that is a block's tail but is not a `JUMP` instruction, therefore it *falls through* to the next block.

Using those predicates, we can define `BlockEdge`. Two blocks are connected by an edge if:

- There is a valid jump from one block to the other
- There is a fallthrough edge between them

Block edge, in turn, allows us to define the reachability relationship, which dictates that a block A is reachable from block B if:

- There is an edge between them
- There is an edge between block B and another block C, and A is reachable from C

With the predicate `MaxFrom` we are able to approximately compute the largest connected code segment in terms of number of blocks.

3.1.5 Unresolved Jumps

Features related to the nature and the number of unresolved jumps would presumably be correlated to the optimal call-site depth. To that end, we compute and collect the following features. The `ThrowJump` relation describes the jump instructions whose jump target is a local variable, but the value of the variable is not a valid destination. A `NonImmediateJump` is a jump performed on a variable that is not local to the block. Moreover, as explained in the previous section, Gigahorse analyzes the local stack by representing stack positions by indices. Non local indices get assigned positive values, while local get assigned negative. We collect a sum of the non immediate and throw jumps, and also compute the highest non-local index that a jump is performed to.

```
ThrowJump(stmt: S)
BasicBlock_Tail(block: B, stmt: S)
IsJump(stmt: S)
BeforeLocalStackContents(stmt: S, idx:number, var: V) \\local stack contents
                                                      \\before statement stmt
CheckIsVariable(var: V)
```

```
C_ThrowJump(n: number)
C_ThrowJump(n):- n = count : ThrowJump(_).

NonImmediateJump(stmt:S, block: B)
NonImmediateJump(stmt, block):-
  BasicBlock_Tail(block, stmt),
  IsJump(stmt),
  BeforeLocalStackContents(stmt, 0, var),
  !CheckIsVariable(var).

C_NonImmediateJump(n: number)
C_NonImmediateJump(n):- n = count : NonImmediateJump(_,_).

JumpVar(stmt:S, var: V)
JumpVar(stmt, var):-
  BasicBlock_Tail(_,stmt),
  IsJump(stmt),
  BeforeLocalStackContents(stmt, 0,var).

MaxJumpVar(n:number)
MaxJumpVar(n):- n = max var : JumpVar(_,var).
```

Figure 3.5: Unresonlved jump predicates

3.1.6 Public Function Related

The public functions of a smart contract are the main entry points to the contract's code. Gigahorse's local analysis is able to find all public function entries in the code by identifying the dispatching patterns discussed earlier. Having the starting blocks of public functions we can compute their approximate sizes in terms of blocks using the reachability predicate introduced in the previous section, as well as approximate gas requirements.

```
PublicFunction(block:B, funHex: signature hash)
```

```
ReachesTo(from: B, n:number)
ReachesTo(from, n):-
    Statement_Block(_, from), n = count : ReachableBlock(from, _).

ApproximatePublicSize(public: B, size: number)
ApproximatePublicSize(public, size):-
    PublicFunction(public, _),
    ReachesTo(public, size).

MaxPublicSize(n:number)
MaxPublicSize(n):-
    n = max size : ApproximatePublicSize(_, size).

ApproximatePublicGas(public: B, gas: number)
ApproximatePublicGas(public, gas):-
  PublicFunction(public,_),
  ReachableBlock(public, to),
  gas = sum block_gas : Block_Gas(to, block_gas).

MaxPublicGas(n:number)
MaxPublicGas(n):-
  n = max gas : ApproximatePublicGas(_,gas).

C_Public(n: number)
C_Public(n):-  n = count : PublicFunction(_,_).
```

Figure 3.6: Public function predicates

3.1.7 Other

Other features considered are in regards to static metrics on various instructions, including:

- Count of total instructions in contract
- Count of `MSTORE/SSTORE` instructions
- Count of `CALLDATALOAD /CALLDATACOPY` instructions

A list of all used features can be found in Appendix A.

3.2 Training Dataset

In order to acquire our training dataset, besides the aforementioned features, we had to decide our dependent variable, i.e the optimal call-site context depth. In this section we describe the approach that was employed for determining that value.

3.2.1 Evaluation Metrics

Finding out the appropriate context-depth for a given contract by examining the source code would be quite a challenging task, as a consequence of EVM's attributes as presented in earlier chapters. In contrast, it is simple to judge the effectiveness of a specific depth on a contract's analysis a posteriori, that is, after the decompilation. That is the main idea behind this approach, effectively assessing the suitability of various depths for each contract in the training dataset based on metrics from the decompilation results. The metrics that were used for this purpose are the following:

- **Polymorphic jump targets** i.e., the number jumps that resolve to at least two targets but have not been resolved as return jumps
- **Missing jump targets** i.e., the number of jumps with unresolved targets
- **Functions with inexact arguments** i.e., the number of functions with uncertain number of arguments
- **Functions with inexact return arguments** same as above, only for return arguments
- **Statements with missing operands** i.e., number of statements of the output IR with unresolved or missing operands
- **Time**, with a timeout limit of 120 seconds

3.2.2 Searching Algorithm

The metrics mentioned allow us to conclude how similar the results of decompilations with different depths are. How we decide which depth is the ideal, though, given the metrics for several depths is not as straightforward as it may seem. For instance, having jumps with missing targets at the result of a decompilation is an indicator of incompleteness, and generally the lower the number of such jumps, the better. It is not unlikely, however, to encounter a decompilation with a higher context depth yielding a result with more missing jump targets than one with a lower context depth. This does not mean that the lower context depth is ideal, since the higher context enables a more accurate value flow analysis, and therefore describes more accurately the actual behaviour of the program.

With all these considerations, it becomes obvious that we are searching for a parameter based on two factors: metrics and time. The optimal context depth for a given contract is the one that has similar metrics with the highest possible context the contract can be analysed with and not timeout, in the least possible time. To calculate this we used the following two-step algorithm.

- First, we calculate the maximum context depth that the contract can be analysed in and not time-out, in a binary search manner. For that binary search we used the maximum depth value of 20, as it was deemed high enough for the vast majority of contracts.
- Then, based on the maximum depth from the previous step, we calculate the depth in the range $[0, max]$ that has the exact same metrics as the maximum depth. If no depth lower than the max has the same metrics, the max is used. This search was also implemented as a binary search.

4. MODEL SELECTION

In this chapter we describe the desired properties of the machine learning model for the specific task of predicting the context parameter and evaluate all of the candidate strategies that were examined.

4.1 The Problem Setting

The dependent variable, the context depth parameter, takes discrete values in a limited range that are also ordered. As a result, it does not fall directly under one of the classic machine learning objectives, since classification algorithms are used to predict finite discrete categorical values with no set ordering, while regression algorithms typically aim to approximate a continuous numerical variable. Furthermore, because of the nature of our variable, the training dataset does not exhibit a balanced distribution among all classes-depths. Contracts that require a low depth are way more common than ones that require a high depth, thus leading to an imbalanced dataset. The created training dataset consists of 23543 unique contracts, exhibiting a distribution of optimal depths as depicted by the histogram in Figure 4.1. As can be seen, the distribution is heavily skewed towards the lower depths. We propose a number of different approaches for tackling the above challenges.

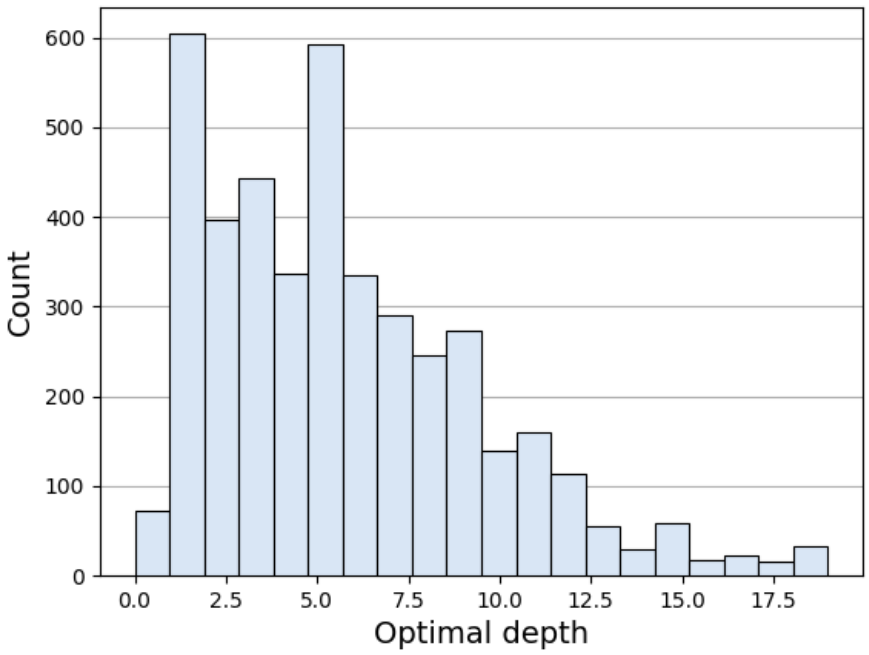

Figure 4.1: Histogram depicting the distribution of optimal depths in the training dataset

4.2 Model Selection

The base estimator for the proposed approaches is the Random Forest [4] (RF). Random Forest is an ensemble-type learner incorporating the bagging algorithm and using Decision

Trees (DT) as weak learners. Ensemble based learners are obtained by training multiple models and combining their results. Empirically, ensemble learners are able to perform better than the best model in the ensemble when there is adequate diversity among the models. The reason behind this is that by achieving diversity each model makes a different error, but the combination of them reduces the total error.

Bagging, also called bootstrap aggregating, is a general ensemble learner, meaning that it can be implemented as an assembly of any type of weak learner. The way bagging ensures model variance is by training each model in the ensemble on a random sample of the training dataset, with replacement. That means that one instance from the dataset can be picked multiple times per sample. Each such sample is called a *bootstrap sample*.

Random Forest uses bootstrap samples for each DT in the ensemble, along with additional random under-sampling of the available features. Using only a subset of the features for each DT it further de-correlates the individual trees. Random Forest, as well as other ensemble learners, perfrom predictions by averaging the predictions of all the models in the ensemble. The property of generalizing with small samples and large amount of features, while exhibiting high accuracy makes RFs an ideal candidate for our prediction task.

The following approaches were evaluated:

- **Random Forest Regression** (Regression RF) with value rounding to the closest integer, using the scikit-learn [9] implementation. This approach has the benefit of intrinsically maintaining the ordinal property of the variable.
- **Balanced Random Forest Classification** (Balanced RF), from the imbalanced-learn library [8] implementation. The bagging algorithm as described earlier, if performed on a heavily imbalanced dataset is likely to produce equally imbalanced bootstrap samples.

  There are two main approaches to sampling for fixing imbalanced datasets; over-sampling and under-sampling. In over-sampling, instances of the minority classes get replicated to match the number of instances of the majority classes. Under-sampling, on the other hand, selects only enough instances from the majority classes to match the minority ones.

  This implementation of RF aims at reducing the side effects of an imbalanced dataset by randomly under-sampling the majority classes in each bootstrap sample.
- **Random Forest with bootstrap class weighting** (Weighted RF), also from the scikit-learn library. The impurity measures used for deciding splits in Decision Trees are influenced directly by the distribution of the classes. Taking the Gini impurity, for example, it calculates the likelihood a randomly selected instance from the training dataset to be misclassified, according to the distribution of the classes. This obviously would lead to a model that strongly favours correctly predicting the majority classes, at the expense of decreased accuracy for the minority.

  Following the idea of cost effective learning, we can influence the impurity measures by penalizing more highly the misclassification of minority classes. The scikit-learn implementation allows us to automatically adjust the weights on each tree depending on the distribution of the separate bootstrap samples, instead of the whole dataset.
- **Ordinal Classification with Random Forest** (Ordinal RF), using an implementation of the algorithm proposed by Frank and Hall [5]. This approach was introduced for

classification problems in which the classes exhibit a natural ordering. The proposed algorithm performs ordinal classification by breaking a k-class ordinal problem in to $k-1$ binary class problems. It works by creating $k-1$ new datasets, having all the same columns as the original, apart from the class column. Each of the $k-1$ datasets has new binary attribute, with the ith attribute representing the property that $class > I$.

For instance, consider we have an ordinal classification task with the class attribute having values in $[1, 2, 3]$. Those class attributes are ordered as such: $1 < 2 < 3$. The new binary attributes for this dataset would therefore be $[class > 1, class > 2]$. In Table 4.1 we show how the class attributes would correspond to the new binary attributes.

Table 4.1: Ordinal classification example

$Class$	$Class > 1$	$Class > 2$
1	$False$	$False$
2	$True$	$False$
3	$True$	$True$

A binary classifier is then trained on each dataset. For this task only classifiers that can return a probability can be used. At prediction time, the probability for each class is calculated based on the $k-1$ probabilities as such:

$$P(1) = 1 - P(> 1)$$
$$P(i) = P(> i-1) - P(> i),\ 1 < i < k$$
$$P(k) = P(> k-1)$$

The instance is then classified according to the class that has the highest probability. The above algorithm was implemented using Random Forest as the binary classifier.

4.3 Model Evaluation

After performing exhaustive search for the tuning of the parameters number of tree estimators and maximum tree depth for each predictor, we evaluated their performance by *stratified 10-fold cross validation*. This evaluation method was chosen because it can closely depict the model's generalization ability. The main idea of k-fold cross validation is to split the training dataset in k equal segments, also known as folds, training the model in $k-1$ of them and evaluating on the remaining one. By performing this evaluation k times, each time selecting different segments for training and evaluation, we are able to evaluate the model on the whole dataset. The term 10-fold cross validation refers to the case the k parameter is set to 10. Lastly, when this algorithm is implemented as stratified, the initial selected segments are chosen so that the class distribution within the samples matches that of the whole dataset.

Because our dependent variable is numeric we evaluated both on numeric metrics such as Mean Absolute Error (MAE) as well as classification metrics like precision, recall and f1. To account for the class imbalances, the metrics precision, recall and f1 were calculated both as macro-averages and micro-averages. To shortly describe the differences between the two, first we need to define the following metrics in the setting of multi-class problems:

- **True Positives for class K** (TP_K) are the instances of the class K that are correctly classified as K
- **False Positives for class K** (FP_K) are the instances wrongly classified as K
- **False Negatives for class K** (FN_K) are the instances of the class K that are misclasified as something else
- **True Negatives for class K** (TN_K) the instances that are not K, classified as something besides K

With those in mind, the general definitions for precision, recall and f1 are as described below:

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \\ F1 &= 2 * \frac{Precision * Recall}{Precision + Recall} \end{aligned}$$

The *micro-average* of precision and recall treats instances of all classes as equal and is indicative of the overall performance of the model on the whole dataset. Hence, they are given by the following formulas:

$$\begin{aligned} Micro\_Precision &= \frac{\sum_{n=1}^{k} TP_n}{\sum_{n=1}^{k} TP_n + \sum_{n=1}^{k} FP_n} \\ Micro\_Recall &= \frac{\sum_{n=1}^{k} TP_n}{\sum_{n=1}^{k} TP_n + \sum_{n=1}^{k} FN_n} \end{aligned}$$

In contrast, the *macro-average* version entails calculating the precision and recall for each class separately and then getting the average for all the classes. As opposed to micro-average, this approach aims at considering all the classes equally, independently of the number of instances in each.

$$\begin{aligned} Macro\_Precision &= \frac{\sum_{n=1}^{k} Precision_n}{k} \\ Macro\_Recall &= \frac{\sum_{n=1}^{k} Recall_n}{k} \end{aligned}$$

Additionally, we defined the extra metric of Desired Accuracy which is the proportion of the predictions of which the error is not larger than 2, as it is a desired property of our model the misclassified instances to not be too far from the optimal value.

Table 4.2: 10-fold cross validation evaluation results

	Weighted RF	Balanced RF	Ordinal RF	Regression RF
MAE	0.544	2.289	0.547	0.865
Accuracy	0.873	0.602	0.868	0.617
Macro_Precision	0.877	0.502	0.873	0.621
Macro_Recall	0.813	0.683	0.806	0.492
Macro_F1	0.842	0.533	0.836	0.513
Micro_Precision	0.871	0.602	0.871	0.618
Micro_Recall	0.871	0.602	0.871	0.618
Micro_F1	0.871	0.602	0.871	0.618
Desired Accuracy	0.927	0.730	0.926	0.893

Table 4.3: Mean fit and score times (sec)

	Weighted RF	Balanced RF	Ordinal RF	Regression RF
Mean Fit Time	5.864	2.055	94.0637	38.067
Mean Score Time	0.102	0.075	2.563	0.103

The results show that the Weighted RF and the Ordinal RF are significantly more accurate than the other two, while yielding almost identical metrics to each other. The results also show that even though the Weighted RF surpasses the Ordinal RF by a slight difference in precision metrics, it exhibits remarkably better fit and scoring times. Overall, the Weighted RF appears to be the optimal choice out of the examined models.

4.4 Feature Importances

Determining which features contribute the most in the predictions is a crucial part of evaluating and interpreting our model. It allows us to verify whether our assumptions about specific feature importances were correct and offers us valuable insights about what is actually correlated with our dependent variable.

There are a few ways of examining that. One of the most commonly used measures of feature importances are the built-in importances that many models offer. For the case of RF, the default feature importances are given in mean decrease in impurity for all the DTs within the RF. Mean decrease in impurity measures how useful a feature is at separating the data and is an indicator of how much a feature is utilized internally in the model. By adding an extra column in our dataset consisting of random numbers we can have a control variable that could demonstrate that any feature with lower importance than that is more or less useless.

Figure 4.2 shows the feature importances as measured by the default mean decrease in Gini impurity by our Weighted RF. The feaure names are replaced by their indices as presented in Appendix A. As expected, the random feature is at the very end, with only one feature being less informative than that. It should be noted, however, that the decrease in impurity has been known to not present the most accurate depiction of importance, as it has been found to favour some types of features over others [11]

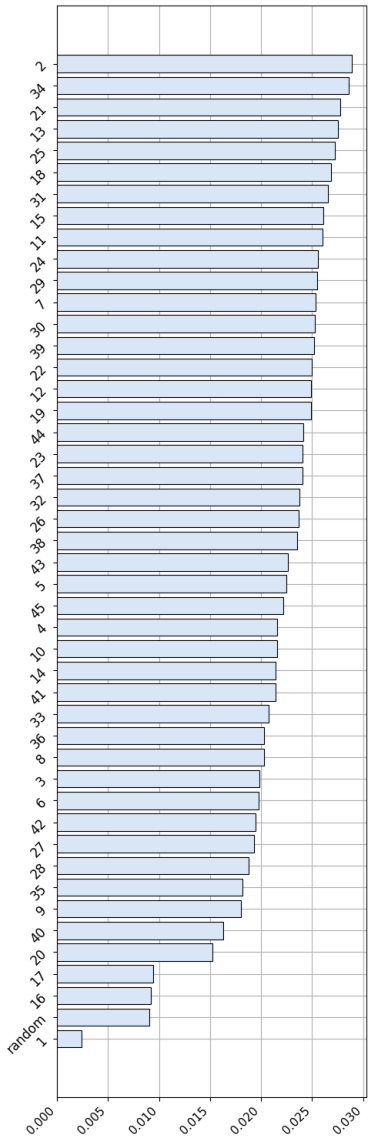

Figure 4.2: Feature importances as measured by mean decrease in Gini impurity

A more impartial method available is what is called the *permutation accuracy importance* measure. The way it is computed is as follows. After training the model with the whole set of features, the accuracy is evaluated and is considered the model's accuracy baseline. Next, for each available feature, the values within the column are randomly permuted and the accuracy of the model is re-evaluated with the new test data. If the feature permuted is important for the model's predictive ability, the difference from the baseline accuracy should be substantial. The features get assigned importances according to the differences in accuracy from the baseline resulting from their permutation.

One common caveat of this approach is that co-linear features could result in non representative importances. Co-linear features can be correlated with a linear or non-linear relationship. By permuting a feature that exhibits co-linearity with another, we will not see an important difference in model accuracy since the model is able to compensate by using the correlated feature. This would result in both the co-linear features getting a low importance measure. A way to overcome this is by grouping and permuting highly correlated features together.

Grouping the features correlated with a Spearman correlation measure higher than 0.85 we can see the permutation importances in Figure 4.3. The results were computed using the

`rfimp` [1] package. Similar as before, the random feature appears in the least important features. Another interesting finding from this computation is that the first seven groups that present the highest importances all seem to share a subset of features.

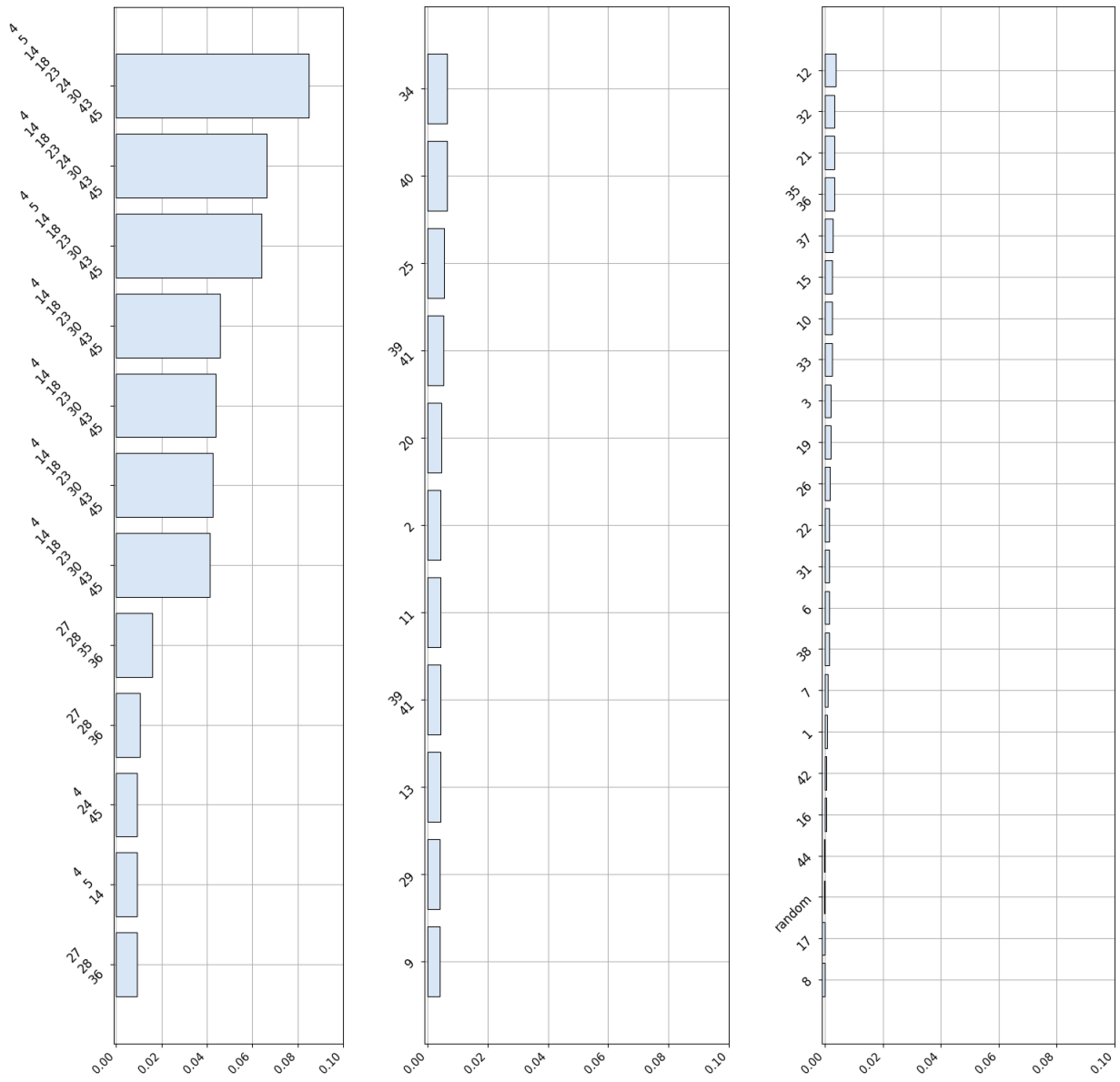

Figure 4.3: Permutation importances, with highly correlated features permuted together

Finally, the effect of the number of features on the f1 and loss scores were examined. For this evaluation, the method of recursive feature elimination was employed. Its rationale is the following: first the model is trained on the initial set of features. Then, based on the model's feature importances, the least important feature is pruned. In figure 4.4 we see a graphical representation of the f1 and loss score according to the number of features selected. The f1 and loss scores stop improving at 40 out of 45 selected features. Despite that, remarkable is the fact that we can achieve scores that are close to the optimal with using only 10 features.

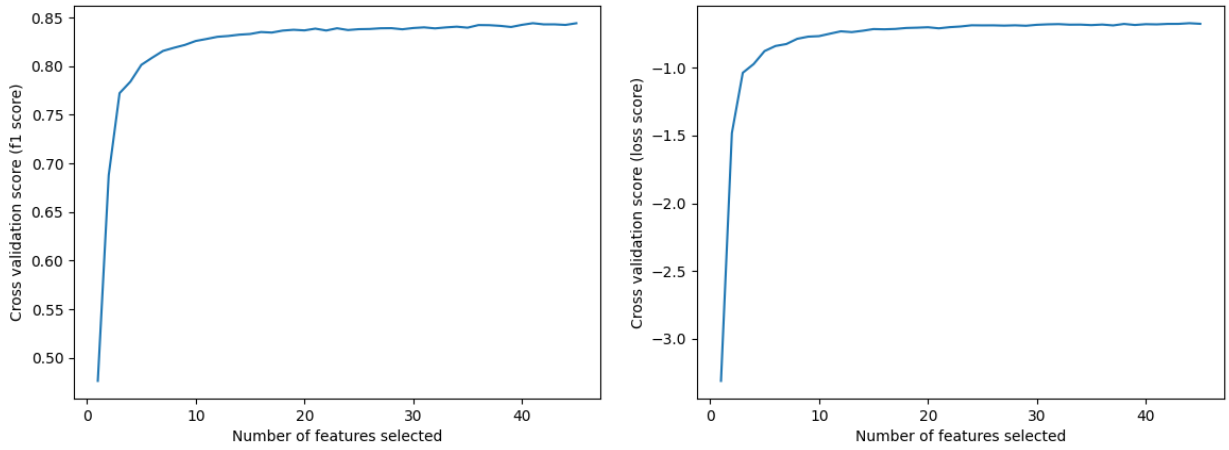

Figure 4.4: F1 and loss scores according to number of features selected

Indeed, when evaluating our Weighted RF model with the top 10 selected features as shown in Table 4.4, we are able to achieve metrics comparable to the ones obtained using the whole set of features.

Table 4.4: Top 10 features selected by recursive feature elimination

Index	Feature
12	MaxPublicSize
13	MaxBlockGas
15	MaxFrom
18	add
24	dup4
30	mstore
31	mul
34	push4
37	revert
39	sload

Table 4.5: 10-fold cross validation evaluation results with 10 selected features

	Weighted RF
MAE	0.620
Accuracy	0.858
Macro_Precision	0.856
Macro_Recall	0.794
Macro_F1	0.822
Micro_Precision	0.857
Micro_Recall	0.857
Micro_F1	0.857
Desired Accuracy	0.918

5. EXPERIMENTAL EVALUATION

In this chapter we evaluate the quality of the predicted depths of our model on a held out dataset. Our test dataset consists of a set of 4235 contracts, randomly sampled from the original dataset, not used in the training phase of our model. To fully demonstrate the differences, we compare the following contexts:

- Predicted depth context (Tuned) using the predictions of our Weighted RF model
- Estimated depth context (Heuristic), obtained by getting the context depth by a heuristic previously used in Gigahorse for depth tuning
- Optimal depth context (Optimal) using the context depths that our analysis, as described in Chapter 3, evaluated as ideal
- Transactional context (Transactional), which is another context flavour used in Gigahorse.

In brief, Transactional tries to qualify with a 2-call-site context blocks belonging in private function calls, along with the additional context of the identifier of the public function performing the call. This aims at increasing the precision of the analysis of private functions while maintaining good scalability. Therefore, we expect it to be generally faster than the other contexts, while at the same time slightly less accurate.

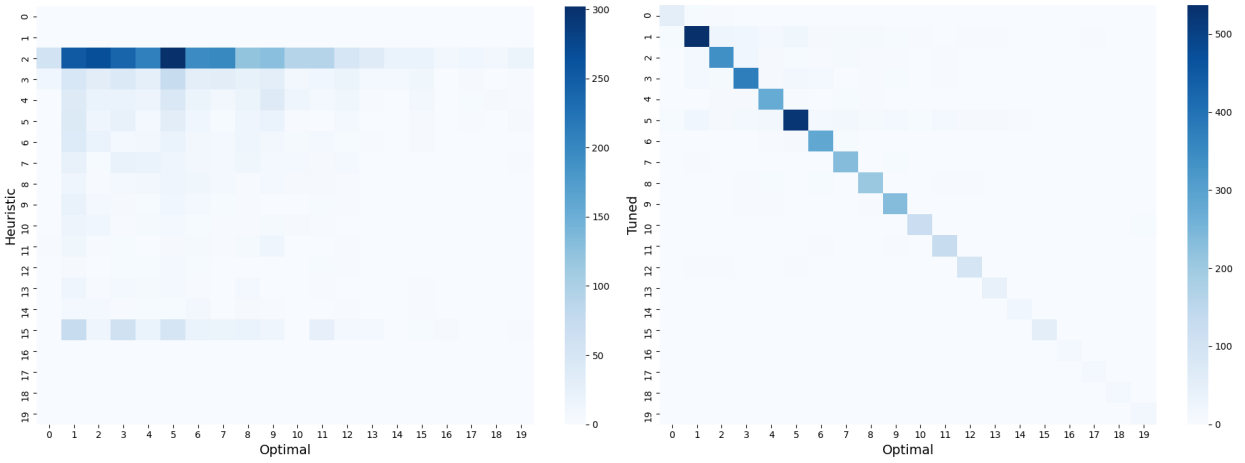

Figure 5.1: Confusion matrices, comparing the Tuned and Heuristic depths to the Optimal depths

In Figure 5.1 we can see two confusion matrices, comparing the Heuristic and the Tuned depth to the Optimal depth. In a confusion matrix, the y-axis represents the predicted classes while the x-axis represents the actual classes. For example, the block positioned in the second row and first column contains the count of instances classified as 2, whose actual value is 1. Consequently, the blocks in the diagonal of the matrix represent the instances that were correctly classified. As Figure 5.1 is meant as a graphical representation, the numerical values inside the blocks were omitted.

As can be seen in Figure 5.1 the Heuristic depth appears to be accumulated mostly in the values 2 and 15. We can notice that the majority of contracts with optimal depth in the $[12, 19]$ range get estimated as 2, while a notable proportion of the contracts with optimal depth in the $[1, 9]$ range get overestimated as 15. It would stand to reason, therefore,

to expect the Heuristic context to generally be less precise for high-demand in terms of context contracts, while leading some in the lower depth group to time out. On the other hand, the predicted depth exhibits a close approximation to the optimal, as exhibited by the confusion matrix.

Table 5.1: Evaluation of decompilations with different contexts

	Tuned	Heuristic	Transactional	Optimal
Timeout percentage	1.6 %	12.7 %	1.2 %	1.0 %
Mean Dec. Time (sec)	15.332	27.407	15.304	14.296
Mean Jump To Many	5.507	9.136	9.368	5.389

In Table 5.1 we can see that the Tuned context has a reduced amount of contracts whose analysis does timeout as compared to the Heuristic, specifically about 8 times less the amount, or an overall reduction of 10 percent. There is also a speedup of the mean decompilation time of 178 percent.

Next on the table we compare the analytic `JumpToMany`. This metric represents the number of blocks that have multiple intra-procedural jump targets in the output functional IR. This differentiates from other polymorphic jumps, for example those encountered in function returns, and only examines the ones that occur inside the scope of a function. The reason this metric is a good measure for the precision of the analysis is the fact that such unresolved jumps usually point to imprecise value flow analysis as they rarely naturally appear in high level code. Typically, it is a safe assumption to make that the lower the number of this kind of jumps, the more accurate the decompilation. Our assumption is confirmed by comparing the `JumpToMany` achieved by the Optimal to the one by the Heuristic. The Tuned exhibits a negligible difference in the metric `JumpToMany` compared to the Optimal.

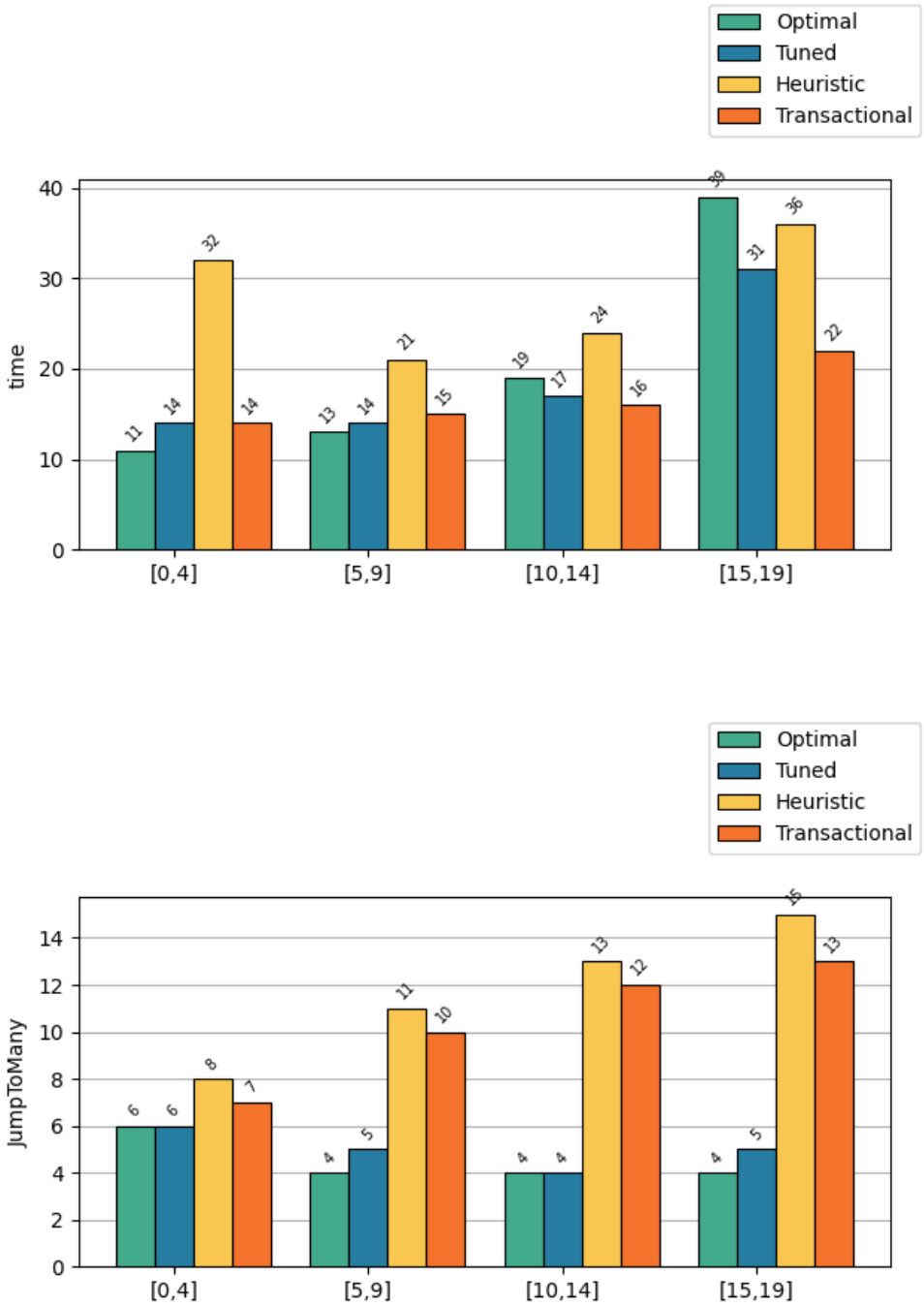

Figure 5.2: Mean decompilation times (sec) and JumpToMany grouped by contacts' optimal depth

By comparing the mean times and `JumpToMany` grouped in categories according to the optimal depth, we get a more clear image. It is apparent that in the lower-depth categories the Heuristic performs much worse in terms of decompilation time, with the other two performing similar to the Optimal. Of interest is the comparison of the `JumpToMany` metric, which highlights our assumptions to be true. The severe loss in precision happens in the higher context groups, with both the Transactional and the Heuristic contexts demonstrating much higher values in that metric. According to Table 5.1, the mean `JumpToMany` of the Optimal is almost half the one achieved by the Transactional and the Heuristic context, while in the high-demand groups, as seen in Figure 5.2, it is close to a third.

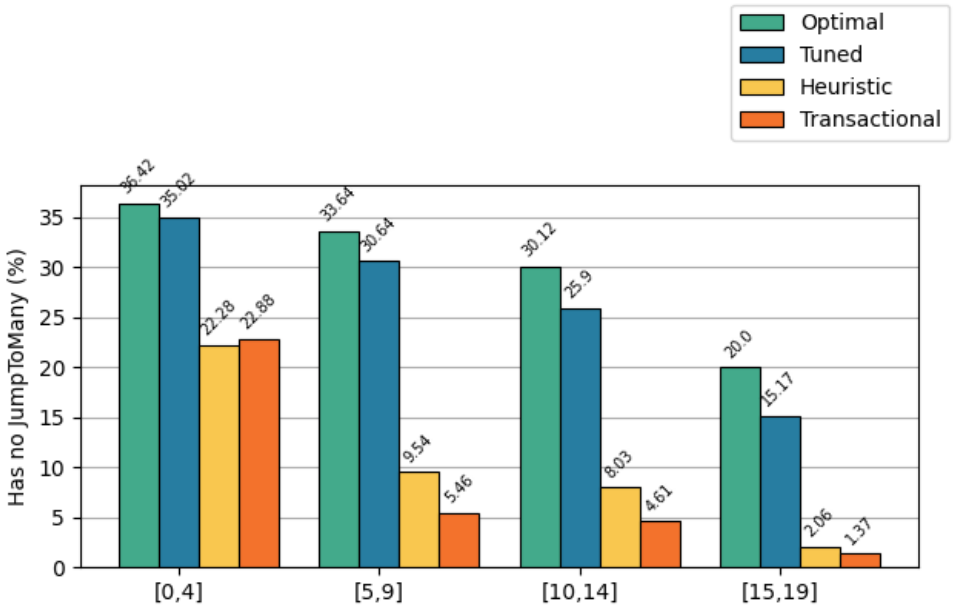

Figure 5.3: Percentage of contracts that have no JumpToMany

To get a better idea of what goes on under the hood with the `JumpToMany` precision metric, in Figure 5.3 we see the percentage of contracts for each different context type that have no `JumpToMany` in the decompilation output. Recall that for this metric, the lower values are preferred. So logically, the higher the percentage of contracts that this metric is zero, the better. The positive effect of the Optimal context becomes apparent, with the Tuned context following a close second.

Following, we compare some more decompilation metrics. Due to nuances in the computation of the rest of the metrics, we cannot ascertain whether generally lower or higher values are ideal. In order to asses the effectiveness of the different contexts we will be comparing the metrics acquired by each context to the metrics acquired by the Optimal context. For this purpose, we calculate the MAE of each metric against the metrics of the Optimal context, to demonstrate the amount of deviation.

Table 5.2: Deviation of decompilation metrics from the metrics of the optimal-depth context

	Tuned	Heuristic	Transactional
Functions	0.733	3.207	0.575
Unknown Operand	0.000	0.018	0.005
Missing Operand	0.245	1.935	1.656
Polymorphic Target	0.112	0.770	0.847
Unreachable Block	6.395	16.538	14.673
Missing Jump Target	4.535	11.323	10.675

From the Table 5.2 we can observe that the Tuned context fares decompilation results close to the ones achieved by the Optimal. The results are even more stark when compared to the Heuristic and the Transactional, both of which generally produce decompilation results that diverge from the Optimal few times more than the Tuned.

6. CONCLUSIONS

To conclude, we presented a machine learning approach that was successful in predicting the appropriate call-site context depth with satisfying accuracy. We were able to decrease the amount of contracts that time out, lower the average decompilation time while at the same time maintaining a good accuracy of the decompilation.

One caveat of the approach is that the way the optimal depth is calculated makes it dependent on the platform, as decompilation times can vary between different machines. In future work, it would be interesting to be explored whether a platform independent model could be constructed.

ABBREVIATIONS - ACRONYMS

EVM	Ethereum Virtual Machine
EOA	Externally Owned Account
IR	Intermediate Representation
ABI	Application Binary Interface
RF	Random Forest
DT	Decision Tree
MAE	Mean Absolute Error

APPENDIX A. FEATURES

	Constructed Features
1	C_ThrowJump
2	C_NonImmediateJump
3	MaxJumpVar
4	C_Instructions
5	C_BlockHead
6	MaxPopDelta
7	C_HighPopDelta
8	C_VeryHighPopDelta
9	MaxStackDelta
10	C_Public
11	MaxPublicGas
12	MaxPublicSize
13	MaxBlockGas
14	TotalGas
15	MaxFrom
16	MaxDupDepth
17	MaxSwapDepth

	Instruction Count Features
18	add
19	and
20	calldatacopy
21	calldataload
22	div
23	dup2
24	dup4
25	dup6
26	exp
27	extcodesize
28	gas
29	gt
30	mstore
31	mul
32	not
33	or
34	push4
35	returndatacopy
36	returndatasize
37	revert
38	sha3
39	sload
40	slt
41	sstore
42	staticcall
43	swap1
44	swap5
45	swap

BIBLIOGRAPHY

[1] Random forest importances.
https://github.com/parrt/random-forest-importances.

[2] Solidity docs.
https://solidity.readthedocs.io/.

[3] A. Antonopoulos and G. Wood. Mastering ethereum. 2018.
https://github.com/ethereumbook/ethereumbook.

[4] L. Breiman. Random forests. 2001.

[5] E. Frank and M. Hall. A simple approach to ordinal classification. 2001.

[6] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. 2019.

[7] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. 2016.

[8] G. Lemaître, F. Nogueira, and C. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. 2011.

[10] Y. Smaragdakis and G. Balatsouras. Pointer analysis. 2015.

[11] C. Strobl, A. Boulesteix, A. Zeileis, and T. Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. 2007.

[12] B. Vitalik. A next-generation smart contract and decentralized application platform. 2013.
https://github.com/ethereum/wiki/wiki/White-Paper.