Decompilation of Ethereum smart contracts

Thomas E. Hybel

Master's Thesis, Computer Science
Advisor: Aslan Askarov
Student number: 201303525
June 2018

ABSTRACT

In this thesis we describe our design and implementation of a decompiler for Ethereum smart contracts, translating from Ethereum Virtual Machine bytecode to a high-level language resembling Solidity. At present no practical decompiler for Ethereum exists, making it infeasible to understand many of the smart contracts on the blockchain. We attempt to solve this problem by adapting traditional techniques and methodology from the decompilation of machine code. Our decompiler works by first translating the bytecode to an intermediate representation, and then applying various data-flow analyses to improve the abstraction level of the program. High-level control-flow structure is also recovered. The program is then converted to an abstract syntax that the decompiler uses to generate high-level code. We have implemented our design in the form of a decompiler called DSol. Our results show that DSol is practical, successfully decompiling and structuring the vast majority of smart contracts taken from the blockchain.

CONTENTS

1 INTRODUCTION

Ethereum smart contracts are programs that run on the Ethereum Virtual Machine (EVM). These smart contracts reside on the Ethereum blockchain in the form of bytecode. The behavior of many contracts is opaque due to a lack of source code, rendering analysis and auditing of these contracts infeasible. We therefore set out to design and implement a decompiler for Ethereum smart contracts, allowing the automatic translation of EVM bytecode to a readable high-level language.

This chapter defines the necessary terminology, explains the problem in more detail, and lists the applications for an Ethereum decompiler.

1.1 TERMINOLOGY

Decompilers

A decompiler is a program that translates programs from a low-level language to a high-level language while preserving the semantics of the translated program [3]. This is illustrated in Figure 1.1. The low-level language could be machine code or bytecode for a virtual machine.

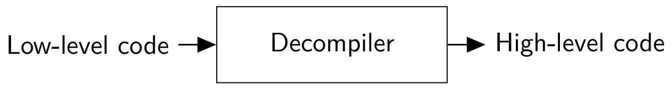


Figure 1.1: The input and output of a decompiler

A decompiler can ease the process of understanding a program when source code is unavailable by lowering the amount of code to read, and by removing machine-specific details from the code.

Blockchain

A blockchain is a tool for achieving global consensus in the absence of a trusted central authority [15]. It can be thought of as a global, append-only data structure. A blockchain stores transactions; a transaction contains data that could, e.g., represent the transfer of currency between two parties.

Cryptographic currencies

A cryptographic currency is a decentralized currency based on cryptography. A blockchain can be used as a ledger, ensuring a global consensus about the ownership of currency at any given time. A plethora of cryptographic currencies exist; some examples are Bitcoin and Ethereum.

The Ethereum project

The Ethereum project features a cryptographic currency called *Ether*, which is the second biggest cryptographic currency after Bitcoin by market capitalization [4]. At the time of writing, Ethereum has a market capitalization of 58898 million USD [9].

Ethereum differs from early cryptographic currencies by additionally providing a platform for so-called smart contracts. An Ethereum smart contract can be thought of as a program capable of sending and receiving Ether. In the words of the Ethereum project website [7]:

> Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference.

The Ethereum Virtual Machine

Ethereum contracts run on the Ethereum Virtual Machine in the form of EVM bytecode. Conceptually, the EVM is a global virtual machine that is available as long as the Ethereum protocol has participants.

Solidity

Solidity is a high-level programming language for Ethereum smart contracts. It is syntactically reminiscent of JavaScript. A program written in Solidity is compiled to EVM bytecode before being run on the EVM.

Ethereum smart contracts

A smart contract has external functions that can be invoked as the result of a transaction, enabling anyone to interact with a contract at any time. When such a transaction is made, participants in the Ethereum protocol execute the smart contract bytecode on the Ethereum Virtual Machine to determine the outcome of the transaction. The outcome could, e.g., be that the smart contract sends money or stores some data in its permanent storage.

Smart contracts have various novel applications. Some examples are a wallet with a daily spending limit; a casino offering a roulette application where tampering with the random number generation is impossible; and decentralized collectibles, such as trading cards.

1.2 PROBLEM STATEMENT

Many smart contracts on the Ethereum blockchain do not have accompanying source code, making their behavior opaque to anyone but the author of the contract.

To solve this problem we propose the design and implementation of a decompiler from EVM bytecode to a high-level language reminiscent of Solidity. The decompiler should produce readable high-level code, making it possible to understand the behavior of smart contracts even when no source code is available. Thus our main research question is:

> Can EVM bytecode be translated into readable high-level code?

We cannot expect to directly use the existing methodology for bytecode decompilation; current bytecode decompilers typically exploit the rich information present in the decompiled binary file, as is, e.g., the case for Java class files. Bytecode decompilers also deal with fundamentally different problems, such as recovering control flow in the presence of exceptions [18].

We also cannot expect to directly use the techniques from machine-code decompilation, since EVM bytecode differs in various ways. We must therefore understand the techniques and methodology used in modern decompilation research, and then pick and adapt parts as needed. This gives rise to the following additional research questions, assuming that our main question can be answered in the positive:

– How can a decompiler for Ethereum smart contracts be designed?
– Which techniques from decompilation research can be applied to Ethereum smart contracts? Which must be adapted?

EVM bytecode differs from machine code by being stack-based, meaning that all instructions take their operands from the program stack. This has two major consequences. First, all jumps are indirect. This makes the reconstruction of a precise control-flow graph difficult. Second, each data access is indirect through the stack pointer, complicating data-flow analyses. Another difference between EVM bytecode and, e.g., x86 machine code, is that in the EVM, inter-function control-flow transfers are ordinary jumps; there is no `call` nor `return` instruction, making function identification difficult. All in all, this gives rise to these additional research questions:

– How can decompilation be achieved in the absence of precise control-flow information?
– How can existing data-flow analyses for decompilation be adapted to handle frequent indirect data accesses?
– How can functions be identified in the absence of `call` and `return` instructions?

We address these issues by using or adapting techniques from the existing research literature on decompilation. When no information exists, we devise our own analyses. After arriving at and describing a design that solves the mentioned problems, we evaluate its practicality by implementing a decompiler that we call DSol.[1] We evaluate our decompiler through experiments that aim to quantify its robustness, correctness, and output readability.

1.3 APPLICATIONS FOR AN ETHEREUM DECOMPILER

A decompiler for Ethereum has numerous applications:

1. Recovering lost source code or contract ABI.

A decompiler eases the recovery of lost source code. It also enables recovery of the application binary interface (ABI) of a contract. Without the ABI, interacting with the contract is impossible. This could, e.g., allow withdrawing the funds from a wallet whose owner lost the ABI.

2. Understanding contracts without public source code

A decompiler makes it feasible to understand contracts even when public source code is unavailable. This ability enables the classification of *every* contract on the Ethereum network, rather than only the ones with available source code. For example, a decompiler can be helpful in answering questions such as "how many of all contracts are wallets?" and "how diverse are contracts on the Ethereum network?"

[1] DSol is pronounced as "diesel": `/'di:z(ə)l/`

3. Enabling vulnerability discovery

Ethereum contracts can suffer from various types of vulnerabilities [1, 5]. A decompiler makes it feasible to understand and search for vulnerabilities in contracts without public source code.

Leaving unethical applications aside, users may still want to audit a smart contract that is not their own in order to minimize the risk of making a bad investment. This argument applies both to individual contracts and to the Ethereum network as a whole, since some smart contracts hold a large percentage of all Ether.[2] If a high-value contract has a vulnerability, it could affect the Ethereum economy as a whole.

In the long term, the public availability of a decompiler may increase the security of the Ethereum network as a whole by making security through obscurity unappealing.

4. Enabling backdoor discovery

The Underhanded Solidity Coding Contest [21] has recently shown a variety of techniques for placing backdoors into contracts, allowing the contract author to steal the funds of investors. Several of these backdoors rely on syntactical techniques that would immediately become obvious from the decompiler output.

Additionally, the compiler could insert backdoors into contracts due to having been tampered with. This tampering could occur anywhere along the path from compiler developer, through software repositories, to the programmer's machine. A decompiler allows the discovery of such an issue.

5. Discovering compiler bugs.

At present, users largely have to trust that the compiler generates correct output, since verifying the low-level code is excessively time-consuming and requires skills that high-level programmers may not have. A decompiler enables users and compiler developers alike to verify the contract bytecode.

6. Understanding how details of smart contracts are implemented at the EVM level.

A decompiler provides a convenient way for users to investigate how high-level language features are implemented at the EVM level.

7. Enabling program analysis at a higher level of abstraction

Some program analyses may only be feasible on a higher-level representation of the program, whether for efficiency reasons or because the decompiler output has more structure than the bytecode.

1.4 STRUCTURE OF THE THESIS

The following chapters cover the theoretical and design aspects of our decompiler. We first provide necessary background information. This is followed by an overview of our design. We then provide an intuition for the process of decompilation through an example. After the overview we describe the theory behind each phase in detail.

[2] For example, the wealthiest contract at the time of writing holds 1% of all Ether, the equivalent of USD 603 million.

After the theory we describe our implementation. This is followed by an evaluation based on a number of experiments, along with a discussion of our results.

The final part of our thesis gives an overview of the related work that we have based our design upon. It also enumerates various tasks that we have left as future work.

2 BACKGROUND

2.1 BLOCKCHAIN FUNDAMENTALS

A blockchain is a protocol for achieving distributed consensus in the absence of a trusted third party [15]. The basic unit in a blockchain is the *transaction*. A block contains a number of transactions, each with some amount of data. A blockchain can thus be thought of as a global append-only log of transactions [36]. A blockchain is so named because it stores transactions in blocks that are concatenated to form one long chain.

Blockchains are relevant to the present thesis because they allow the implementation of cryptographic currencies. A cryptographic currency, or *cryptocurrency* for short, is a decentralized virtual currency based on public-key cryptography [36, 15]. In a cryptocurrency, the blockchain functions as a global ledger that keeps track of how much currency belongs to whom at a given time. Currency can be transferred to another party by signing a transfer with one's private key.

The primary obstacle in realizing a cryptographic currency is the so-called double spending problem, where a malicious actor spends their money twice. To illustrate the double spending problem, imagine three actors: Alice, Bob, and Eve. Eve signs a transaction saying that she transfers all her money to Alice. After receiving the paid-for goods from Alice, Eve then signs another transaction saying that she sends all her money to Bob, receiving more goods in return. This is the double spending problem.

The issue in this example is that transactions are not global; Bob is unaware of the transaction from Eve to Alice. A blockchain solves this problem by functioning as a global distributed ledger: Eve appends her first transaction to the chain, and now all parties agree that the money belongs to Alice.

In a blockchain, each block B_i holds a number of transactions. The block B_i also holds a cryptographic hash of the previous block B_{i-1} to which it was appended. This ensures that if B_{i-1} is ever replaced with some other block B'_{i-1}, then all blocks thereafter will be invalidated since the hash will no longer match. This ensures that the blocks conceptually form a chain.

To create a valid block, a cryptographic proof of work must be solved. As an example of a proof of work, consider the problem of appending some number n to the block such that the SHA-256 hash of the block has k leading zeros. Such a proof of work ensures that creating a valid block is computationally expensive. Ethereum uses a much more complex proof of work called Ethash[1] [23], although the principle remains the same.

Blocks with valid proofs of work are computed by so-called miners; a miner refers to a machine (or sometimes the owner of said machine) that collects transactions from users, concatenates them into a block, and attempts to solve the proof of work for the block. Miners are connected in a peer-to-peer network, with node discovery inspired by the Kademlia protocol [37, 44]. Computing a valid proof of work is called mining the block. While mining a

[1] Ethash is designed to prevent miners from using application-specific integrated circuits (ASICs) as specialized hardware. This ensures that mining remains profitable for individuals, with the result of increasing the diversity of the Ethereum network. Ethash accomplishes this by being memory-bound rather than CPU-bound.

block is costly in terms of CPU power, the miner that succeeds is compensated by being awarded some amount of currency.[2] Miners are thus incentivized to partake in the protocol, keeping the system running.

If two valid blocks are mined, both of which are valid successors to a previous block, then the chain is said to fork, splitting into two chains. In that case, only one of the two chains is valid, namely the one that is "longest", meaning that it required the most computing power to create.

Returning to the double spending problem, the only way Eve can spend her money twice is if she goes back to the block B_{i-1} where she still held her money, and creates a new successor block B'_i in which her money goes to Bob instead of Alice – that is, by forking the chain. For the new chain to become the globally accepted one, Eve must make her new chain longer than the old one; but if Eve holds less computing power than the rest of the miners, the original chain continues to grow faster than Eve's, making double spending impossible. An important security assumption is therefore that miners with a majority of the computing power do not collude.

2.2 ETHEREUM SMART CONTRACTS

Ethereum is one example of a cryptographic currency. It is different from traditional cryptographic currencies because it includes the concept of smart contracts.

An Ethereum smart contract is a program for the Ethereum Virtual Machine. A smart contract is capable of sending and receiving currency. A smart contract has callable functions; external functions can be invoked by anyone at any time by creating a transaction to the contract address.

Ethereum smart contracts are represented as bytecode for the Ethereum Virtual Machine. The contract bytecode is stored on the blockchain, providing strong integrity guarantees.[3] The bytecode for a contract can never be changed after contract creation.[4]

When a transaction to a smart contract is included in a block, *every* miner trying to mine the block executes the contract bytecode on the Ethereum Virtual Machine to determine the outcome of the transaction. If a miner does not follow the specification of the EVM to the letter, then other miners will notice this fact and declare the mined block void, which means that the miner does not receive any compensation. The result is that the EVM can be regarded as a trusted computing platform, at least as long as the security assumptions on the Ethereum protocol hold.

The fact that each miner runs each transaction on their machine has security implications. First, it means that the EVM must provide isolation from the rest of the system. Additionally, it means that denial-of-service attacks are possible. For example, if a transaction to a contract results in an infinite loop, the miner will spend computing power futilely. To address this problem, the initiator of a transaction pays the miners for every computational step. The

[2] This happens without any third party awarding the compensation; instead, the successful miner is allowed to insert a transaction of currency to themselves into the block. This is a cause of inflation.

[3] Actually the bytecode itself is not stored on-chain for space efficiency reasons. Instead, a hash of the bytecode is stored on the blockchain, while the bytecode itself is stored in a peer-to-peer network called Swarm.

[4] This has the consequence that if a contract vulnerability, contract bug, or compiler bug is discovered after contract deployment, there is no way to fix it! Therefore, in practice, smart contracts tend to contain a "backdoor" that lets the owner move its money to a different account, so that a fixed version of the contract can be redeployed. Unfortunately this removes one of the main features of a smart contract, namely that only the code needs to be trusted, not its owner.

result is that a denial-of-service attack is prohibitively expensive. In slightly more detail, the initiator of a transaction pays for a pool of so-called *gas* before the transaction is processed. Every EVM instruction has an associated gas cost, which is subtracted from the pool of gas as the miner processes the instruction. If the pool runs dry, all effects of the transaction are reverted. Either way, the amount paid for this gas is transferred to the miner as an additional incentive for mining.[5]

Smart contracts are useful for much the same reasons as a traditional contract is: it provides a binding, enforceable agreement between parties, e.g., regarding matters of currency. Unlike a normal contract, a smart contract is written in a programming language and is cryptographically enforced. An example of a smart contract is a wallet with a daily spending limit. Another example is a casino offering a roulette application; since the code is public, users can verify for themselves that random number generation is fair.[6]

Ethereum smart contracts are typically written in a high-level language and compiled to EVM bytecode. One popular smart contract programming language is Solidity.

2.3 SOLIDITY

Solidity is a high-level smart contract programming language. Syntactically it is reminiscent of JavaScript. Solidity also inherits its scoping rules from JavaScript, such that variable declarations are hoisted to the beginning of the function [47]. Unlike JavaScript, however, Solidity is statically typed.

A Solidity smart contract has a number of functions, instance variables, and a constructor. Functions may be externally invokable, in which case anyone can do so as the result of a transaction. An example of a smart contract written in Solidity is given in Figure 2.1.

```
contract SmallExample {

    function myfunc (uint x) external returns (uint) {
        uint result = 0;

        if ((x % 2) == 1) {
            result += 1;
        }
        else {
            result += 2;
        }

        for (uint i = 0; i < x; i++) {
            result += 2;
        }

        return result/3 + 5;
    }
}
```

Figure 2.1: An example smart contract written in Solidity

[5] Since miners are free to choose which transactions to include in the block they are attempting to mine, this provides a way to pay extra to have a transaction processed sooner: by providing additional gas for a transaction, miners will presumably preferentially include that transaction in their block.

[6] However, since the Ethereum Virtual Machine is deterministic, secure random number generation is notoriously difficult.

Functions in Solidity can potentially have multiple return values. Solidity also supports various complex data types, including strings, arrays, mappings, and structs.

An Ethereum smart contract has an area of memory called storage; this is where its instance variables reside. Data in storage persists across transactions. This contract storage can be thought of as residing on the blockchain, in the sense that it is integrity-protected. However, data in storage is not explicitly represented on the blockchain – rather, the transactions that led to the program code modifying storage are stored on the chain. To compute a current view of contract storage, miners must replay all past transactions pertaining to the contract.

2.4 THE ETHEREUM VIRTUAL MACHINE

Ethereum smart contracts are represented on the blockchain in the form of EVM bytecode. The EVM is a stack-based virtual machine: the operands of instructions are popped from the stack, and the result of a computation is pushed onto the stack as a result of executing an instruction. The EVM is big-endian with a large word size of 256 bits. This is, e.g., convenient for computations on large values such as SHA-3 hashes and account addresses, but inefficient for computations on small values.

Instruction set

The EVM instruction set is similar to other stack-based VMs, such as the Java Virtual Machine, with most instructions dedicated to performing arithmetic and managing the program stack. An example snippet of EVM instructions is provided in Figure 2.2. The example code jumps to address `0x44` if it holds that `(1+2)*3 == 9`.

```
PUSH 0x1
PUSH 0x2
ADD
PUSH 0x3
MUL
PUSH 0x9
EQ
PUSH 0x44
JUMPI
```

Figure 2.2: An example snippet of EVM instructions

The instructions for managing the program stack are `POP`, `PUSH`, `SWAP` and `DUP`.[7] There is a variety of arithmetic instructions, such as `ADD`, `MUL` and `XOR`.

Control-flow transfers happen through the `JUMP` instruction. This includes function calls and returns – there is no separate instruction to call a function, nor to return from one. Conditional jumps occur through the `JUMPI` ("jump if not zero") instruction. The `JUMP` and `JUMPI` instructions take their destination argument on the program stack, which means that every jump is indirect.

[7] The latter three are actually *classes* of instructions; EVM has various `PUSH` instructions depending on the data width, such as `PUSH1`, `PUSH2`, etc. up to `PUSH32`. Similarly, there are various `SWAP` instructions depending on which stack word is swapped; thus there is a `SWAP1` instruction, a `SWAP2` instruction, and so on. We have grouped these for simplicity. In our implementation, the class of `DUP` instructions is treated as a single instruction with an operand, which is the stack offset to duplicate. The other instruction classes are treated similarly.

The consequence is that exact control-flow information is not available except through extensive analysis. Jumps may only target destinations marked with a `JUMPDEST` instruction in the code; jumps elsewhere cause the transaction to be reverted.

The EVM also has a number of instructions specific to smart contracts. To give some examples: the `BALANCE` instruction can be used to retrieve the current balance of an account. The `CREATE` instruction creates a new smart contract. The confusingly named `CALL` instruction invokes code on another contract – it does *not* call a function on the current contract. Similarly confusing is the `RETURN` instruction; it does not return from a function call, but rather halts execution altogether, marking an area of memory as the result of the transaction.

Memory architecture

The EVM has a complicated memory architecture, with a large number of dedicated memory regions, including regions for the program stack, memory, storage, call data, program code, and return data. An explanation of each memory region is given in Figure 2.3.

Name	**Description**	**Instructions**
Stack	A region which holds instruction operands and results. Modified when processing most instructions.	`PUSH`, `POP`, `SWAP`, `DUP`
Memory	A byte-addressed region that is used as a scratch space. It is erased after each transaction. Cheaper to use than storage.	`MLOAD`, `MSTORE`, `MSTORE8`
Storage	A word-addressed region that holds the instance variables of a contract. Persists across transactions.	`SLOAD`, `SSTORE`
Code	A read-only region holding the program code.	`CODESIZE`, `CODECOPY`
Call data	A read-only region holding the packed arguments of an external function call.	`CALLDATASIZE`, `CALLDATALOAD`

Figure 2.3: The EVM memory regions

Deployment bytecode

When a smart contract is created, its bytecode is *not* put directly on the blockchain. Instead, each contract goes through a deployment process. What *is* placed on the blockchain is so-called deployment bytecode. The miners execute this deployment bytecode as a result of the contract creation transaction. When the deployment bytecode halts, it returns the actual contract bytecode that is henceforth used to process any transactions. If a contract has a constructor, the bytecode for the constructor is included in the deployment contract. Thus the work of the constructor is performed while executing the deployment bytecode, ensuring that the constructor is run during contract creation and never thereafter.

This has consequences for decompilation: it means that the contract constructor must be recovered from the deployment code, as it no longer exists once the contract is deployed. It also means that the deployed bytecode must be extracted from the deployment bytecode as part of the decompilation process.

3 OVERALL DESIGN

This chapter gives an overview of the design of our decompiler. We provide an intuition for what the various stages of the decompiler do, leaving the details for later chapters. We also explain the motivations behind our design decisions.

3.1 DESIGN CONSIDERATIONS

Correctness

We consider the correctness of our decompiler crucial. That is, the output must be semantically equivalent to the input bytecode. One of the main uses of a decompiler is to ease the understanding of programs. If this understanding is wrong, however, then the utility is lost, and the analyst can no longer trust the tool. We therefore consider correctness one of the top priorities in our design.

For our design, this means that our analyses must always make the safe choice when there is doubt. It also means that we must minimize the use of heuristics that only produce correct results some of the time. In addition, our implementation should check as many invariants as possible to increase the chances of catching incorrect output. We prefer to produce no output at all over producing incorrect output.

Output compilability

It is preferable that a decompiler produces output that is syntactically correct and complete, so that it can be compiled. However the main intended use of our decompiler is to let humans, not the compiler, understand programs. Demanding that the output be compilable also makes our task significantly more difficult, as *every* low-level detail must then be removed. Therefore we choose to not make output compilability a priority.

Performance

Our decompiler should be useful in practice. This means that it should have a practical running time for all realistic contracts; that is, the running time on an average contract should be on the order of minutes, not hours or days. As long as this requirement is satisfied, we do not put much additional value in faster running times; performance is therefore of low priority in our design. As a consequence, we prefer algorithms and techniques that are easy to implement over ones that are highly performant.

Readability

Output readability is important because it facilitates the main intended use of our decompiler, namely understanding programs. However readability should never come at the cost of correctness.

Modularity

Our design is made with a single purpose in mind: to translate from EVM bytecode to a Solidity-like language. To facilitate achieving this, we have deliberately *not* made modularity a priority. In other words, our design is shaped around the peculiarities of EVM bytecode and the Solidity language. Our design may thus require severe changes to support other architectures.

Supported inputs

We assume that the input for our decompiler was produced by a compiler. It is possible to write a contract manually using EVM assembly. It is also possible to obfuscate bytecode. We do not make it a priority to handle such inputs.

3.2 DESIGN OVERVIEW

For our design, we use the ideas of Cifuentes [3] as our starting point. We elaborate on the work of Cifuentes in Section 15.2. For now, this means that our design is divided into three stages: a front end, a middle end, and a back end. This is illustrated in Figure 3.1.

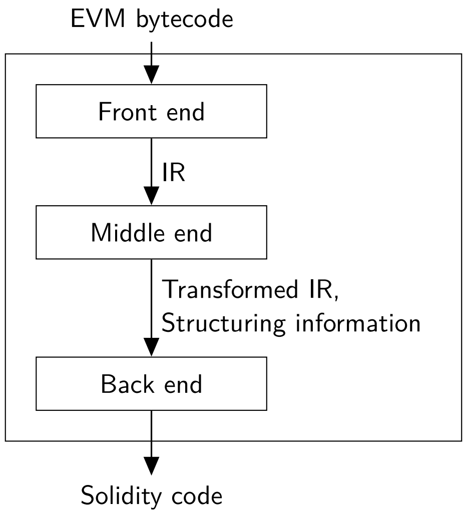


Figure 3.1: Overall design of the decompiler.

The front end converts the input bytecode to an intermediate representation. The middle end recovers high-level concepts through data-flow and control-flow analyses. The back end produces high-level code.

3.3 FRONT END

The front end takes EVM bytecode as its input. It transforms the bytecode into an intermediate representation (IR) suitable for the middle end to work upon.

The front-end has two stages. First is a parsing stage that splits the EVM bytecode into separate EVM instructions and divides these into basic blocks. Then comes a conversion phase that converts the EVM instructions to an intermediate representation. This is illustrated in Figure 3.2.

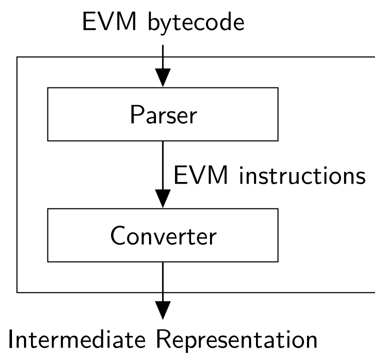


Figure 3.2: Design of the front end.

3.4 INTERMEDIATE REPRESENTATION

In our final design, we have chosen to represent each function in the program as a control-flow graph (CFG) that consists of basic blocks interconnected by edges. Each basic block contains a sequence of statements, where the last statement transfers control flow to another basic block or terminates the program. These statements have optional arguments and results, all of which are expressions.

The edges in the CFG represent possible flows of control. When an exact successor cannot be determined, e.g., in case of an indirect jump, then we use a sound over-approximation of these possible flows. In other words, basic blocks with indirect jumps have edges leading to *every* basic block. The precision of the CFG is later increased by various analyses in the middle end.

We chose this intermediate representation, with basic blocks connected in a CFG, because it is a common representation in decompiler design (see, e.g., Cifuentes [3]; van Emmerik [22]; and Yakdan et al. [25]).

At one point we experimented with changing our intermediate representation to a more complex one, with various node types representing high-level language constructs. That is, there was a `Sequence` of statements, a `Loop` node, a `IfElse` node, and an `IndirectJump` node. The middle end iterates some data-flow analyses to a fixed point; we hoped that our inclusion of a `Loop` node would allow the inclusion of control-flow analyses into this fixed-point computation, resulting in more readable output.

Unfortunately this representation turned out to be cumbersome; every one of our analyses had to case over and explicitly handle every node type, which complicated our analyses and made their implementation error-prone. We therefore abandoned that intermediate representation and reverted to the simpler one with only basic blocks.

Statements

The basic blocks of our intermediate representation contain statements which operate upon expressions. Our choice of statement types largely followed from the input and output languages. For example, Solidity has an `assert` statement, so a statement of this type is natural to include in our representation.

The most important types of statements are `assign`, `jump`, `jcond`, and `vm-call`. The `assign` statement assigns an arbitrarily complex expression to a variable. An example `assign` statement is:

```
var0 := (var0 + 2)
```

The `jump` statement is self-explanatory, while `jcond` is a jump that is conditional based on an expression. The `vmcall` instruction invokes EVM-specific functionality, e.g., causing the machine to halt. Later on, analyses in the middle end create statements with types that do not exist in EVM bytecode, such as `call`, `return` and `assert`.

We considered using static single-assignment (SSA) form in our intermediate representation. SSA form is a property of an intermediate representation. In SSA form, every variable in the program is assigned to exactly once [29]. This simplifies certain data-flow analyses because every use of a variable is associated with a unique definition. For these and other reasons, van Emmerik [22] argues that SSA form is advantageous in machine-code decompilation. However, converting to and maintaining this form introduces additional work. Additionally, indirect memory operations, which are common in EVM bytecode, complicate the conversion and maintenance of SSA form [33]. We therefore decided not to use SSA form.

Expressions

The statements of our intermediate representation operate upon expressions. The types of these expressions follow from the instructions and memory architecture of the EVM. For example, EVM bytecode has an `ADD` instruction, so our representation needs a binary operator representing addition. As another example, the EVM has a memory area called storage. Our representation therefore has a storage variable type.

The types of expressions are divided into literals, unary- and binary operators, and variables. Some example expressions are provided in Figure 3.3.

Example expressions:

```
1
1 + 2
~(2 ** (256 - 32))
var0
mem[var0:(stack[sp] - var0)]
```

Example variables:

```
var0
globalvar1
stack[sp+1]
mem[0x20:0x40]
storage(0x0)
```

Figure 3.3: Examples of expressions and variables

After the front end has converted the input bytecode to our intermediate representation, the program is passed on to the middle end.

3.5 MIDDLE END

The purpose of the middle end is to remove low-level details from the code, transforming it to a higher level of abstraction that is more suitable for generating high-level code.

The analyses included in the middle end are chosen based on the low-level details that must be removed and the high-level details that must be restored to produce readable high-level code. For example, one of these analyses is expression propagation, which restores complex expressions from the simple ones that the EVM can compute. As a result of expression propagation, some definitions become unnecessary. We therefore include dead-code elimination in our list of analyses. The middle end must also recover functions, which do not exist anymore after compilation to EVM bytecode.

One central issue when decompiling EVM bytecode is that a precise control-flow graph is not immediately available, since all jumps are initially indirect. This leads to the following problem. Our data-flow analyses can improve the precision of the CFG, but these analyses are in turn more effective with a more precise CFG. This paradoxical situation is an inherent part of analyzing binary programs with indirect jumps [40], but it is a particularly big problem for EVM bytecode since *every* jump is indirect.

To resolve this problem, our decompiler repeatedly applies the analyses until reaching a fixed point. This gradually increases the precision of the CFG. To the best of our knowledge, this is a deviation from previous decompiler designs, which apply each analysis in the middle end only once. Note that the benefits of such a fixed-point iteration may be higher for decompiling EVM bytecode than for traditional machine code, since the former has greater inherent imprecision.

Once a fixed point is reached, a control-flow analysis phase is run once, recovering loops and conditionals. We do not include the control-flow analysis in the fixed-point computation because when we did so, it complicated our intermediate representation and the implementation of every other analysis, as described in Section 3.4.

Our design, based on these considerations, is illustrated in Figure 3.4.

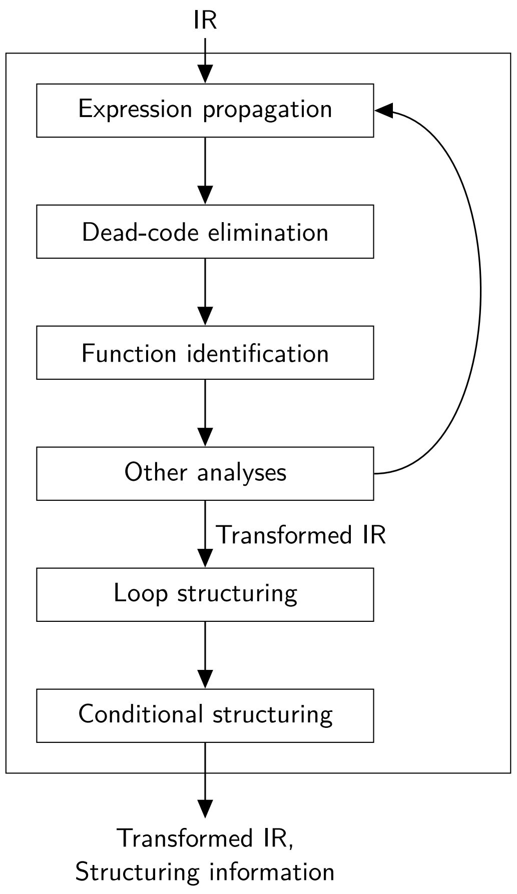


Figure 3.4: Design of the middle end.

The following chapters describe each analysis in more detail. For now we provide a brief overview of what each analysis accomplishes.

The data-flow analyses in the middle end include expression propagation and dead-code elimination. In expression propagation, the definition of a variable is propagated to its point of use whenever this is safe. This turns the simple sequential operations of EVM bytecode into higher-level expressions. As a result of such propagations, the initial definition may become unused. Dead-code elimination removes such definitions.

Function identification involves splitting code off into separate functions when it is safe to do so. Since EVM has no `call` instruction, there is no trivial way to discover functions; therefore we initially treat all of the code as one large function starting at address `0x0`. This includes the loader code and all other code as well. It is the task of the function identification module to detect jumps that are actually function calls. When a function is identified, its code is split off for separate analysis.

The control-flow analyses include structuring of loops and structuring of conditionals. In loop structuring, loops and their contained basic blocks are discovered. Structuring conditionals involves finding the so-called follow node of a conditional jump; the follow node is the point where the two branches meet. These analyses makes it possible to generate more readable high-level code that contains no `goto` statements.

3.6 BACK END

The back end receives the transformed intermediate representation from the fixed-point iteration in the middle end, along with structuring information from the control-flow analyses. The purpose of the back end is to generate high-level code in a Solidity-like programming language.

The back end has three stages: abstract syntax tree (AST) conversion, readability improvements, and code generation. The design of the back end is illustrated in Figure 3.5.

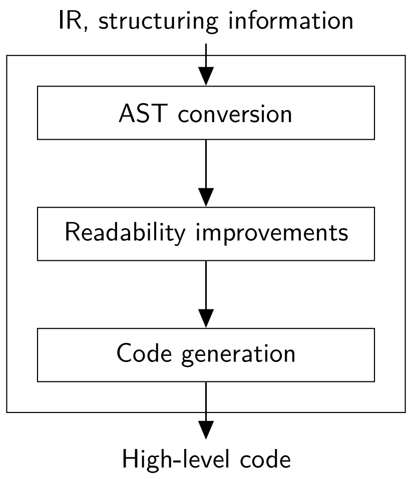


Figure 3.5: Design of the back end.

In the first stage of the back end, the intermediate representation of each function is transformed into an abstract syntax tree. The structuring information is used to accomplish this. Node types include `IfElse`, `IndirectJump`, `Loop`, and `Sequence`.

In the second stage of the back end, various passes are made over the abstract syntax, with the goal of increasing the readability of the final code. These passes include variable naming, identifying array accesses, identifying casts to smaller types, and other minor transformations.

In the third and final stage, a pass is made over each AST to generate the code for a function. This involves recursively visiting each node in the function and generating the appropriate code. The process is rather straightforward once the program is in AST form.

Occasionally the middle end may be unable to remove all the low-level details from the intermediate representation. For instance, some indirect jumps may remain due to expression propagation or function identification failing. In that case code generation produces `goto` statements that do not exist in Solidity. This means that the code is not compilable without manual changes. In effect, we are generating code in a Solidity-like language; the output is only valid Solidity in the ideal case.

4 DECOMPILATION BY EXAMPLE

To give an intuition for what each stage of the decompiler does, this section illustrates the inputs and outputs of each phase of our decompiler as it processes a small example contract.

Solidity source code

Our running example is based on the Solidity smart contract called `SmallExample`. It has the following source code:

```
contract SmallExample {

    function myfunc (uint x) external returns (uint) {
        uint result = 0;

        if ((x % 2) == 1) {
            result += 1;
        }
        else {
            result += 2;
        }

        for (uint i = 0; i < x; i++) {
            result += 2;
        }

        return result/3 + 5;
    }
}
```

The smart contract has a single externally callable function, `myfunc`, which performs some arithmetic computations using an `if` statement and a `for` loop for illustration purposes.

Bytecode

Compiling the smart contract gives the following bytecode, represented as a hexadecimal-encoded string:[1]

```
60606040526004361060 3e5763ffffffff7c0100000000000000000000000
0000000000000000000000000000000000600035041663acc9d5d6811460
43575b600080fd5b3415604d57600080fd5b6056600435606856 5b604051
908152602001604051809103 90f35b600080806002840660011415608157
600182019150608856 5b600282019150 5b5060005b838110 1560a1576002
919091019060010160 8c565b506003900460050192915050 5600a165627a
7a72305820 8463eb00770034296b7ef4643451d56dceb2323d346540025e
4d14069220c6880029
```

This bytecode is given as input to the front end of the decompiler, which has two stages: parsing and conversion to an intermediate representation.

[1] The compilation actually produces *deployment bytecode* that, when executed, produces the shown bytecode; the details are explained later.

EVM instructions

After parsing, the program is represented as basic blocks containing EVM instructions:

```
0x0:
PUSH 0x60
PUSH 0x40
MSTORE
PUSH 0x4
CALLDATASIZE
LT
PUSH 0x3e
JUMPI

0xc:
PUSH 0xffffffff
PUSH 0x100..000
PUSH 0x0
CALLDATALOAD
DIV
AND
PUSH 0xacc9d5d6
DUP 0x2
EQ
PUSH 0x43
JUMPI

0x3e:
JUMPDEST
PUSH 0x0
DUP 0x1
REVERT

0x43:
JUMPDEST
CALLVALUE
ISZERO
PUSH 0x4d
JUMPI

0x49:
PUSH 0x0
DUP 0x1
REVERT

0x4d:
JUMPDEST
PUSH 0x56
PUSH 0x4
CALLDATALOAD
PUSH 0x68
JUMP

0x56:
JUMPDEST
PUSH 0x40
MLOAD
SWAP 0x1
DUP 0x2
MSTORE
PUSH 0x20
ADD
PUSH 0x40
MLOAD
DUP 0x1
SWAP 0x2
SUB
SWAP 0x1
RETURN

0x68:
JUMPDEST
PUSH 0x0
DUP 0x1
DUP 0x1
PUSH 0x2
DUP 0x5
MOD
PUSH 0x1
EQ
ISZERO
PUSH 0x81
JUMPI

0x78:
PUSH 0x1
DUP 0x3
ADD
SWAP 0x2
POP
PUSH 0x88
JUMP

0x81:
JUMPDEST
PUSH 0x2
DUP 0x3
ADD
SWAP 0x2
POP

0x88:
JUMPDEST
POP
PUSH 0x0

0x8c:
JUMPDEST
DUP 0x4
DUP 0x2
LT
ISZERO
PUSH 0xa1
JUMPI

0x94:
PUSH 0x2
SWAP 0x2
SWAP 0x1
SWAP 0x2
ADD
SWAP 0x1
PUSH 0x1
ADD
PUSH 0x8c
JUMP

0xa1:
JUMPDEST
POP
PUSH 0x3
SWAP 0x1
DIV
PUSH 0x5
ADD
SWAP 0x3
SWAP 0x2
POP
POP
JUMP

0xaf:
STOP
```

Address 0x0 contains loader code, which is responsible for dispatching to the called function. For our example, the loader code is unneeded because there is only a single external function, but the compiler produces it nonetheless.

The loader code checks if the first four bytes of the input are 0xacc9d5d6, which is a hash of the myfunc function signature. If so, the loader code calls myfunc. Worth noting is that the basic block at address 0xa1 computes result/3 + 5 and thus belongs to the myfunc function.

Conversion to IR

The next stage of the front end involves the conversion of EVM instructions into an intermediate representation.

Conversion of most arithmetic instructions is straightforward since our IR can represent stack variables. For example, the EVM instruction ADD pops two words from the stack, adds them, and pushes the result to the stack. An ADD instruction can therefore be converted to the assignments:

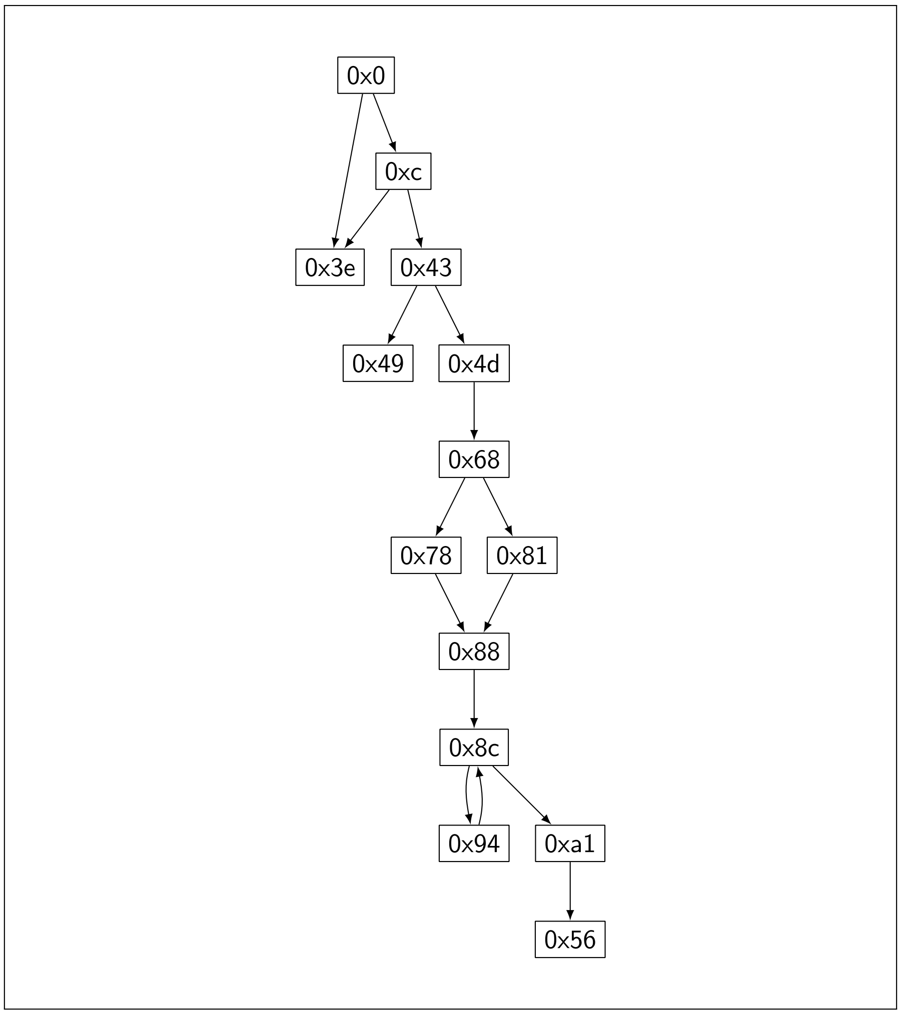
```
stack[sp-1] := stack[sp-1] + stack[sp]
sp := sp - 1
```

where sp is a global variable representing the stack pointer. The concepts of a stack and a stack pointer are ideally eliminated at a later stage, as they do not exist in Solidity.

Immediately after conversion, the whole program is represented as a single large function. This changes once function identification runs in the middle end. All changes to the `sp` variable in each basic block are consolidated into a single change block as part of the conversion process.

While all jumps are indirect in EVM, the conversion module can produce a reasonably accurate initial CFG by recognizing the `PUSH literal; JUMP` pattern.

The CFG of the function that represents the program after conversion looks as follows, where we have left out instructions for space reasons:



Immediately after conversion, the IR of the basic blocks can be difficult to read, since values are copied excessively between variables as a result of the conversion process. This is improved after data-flow analysis. As an example, consider the basic block at address `0xa1`, which computes `result/3 + 5`. After conversion, the basic block contains the following statements:

```
sp += -4                          // consolidated sp-delta is -4
var(0) := stack[sp+4]             // POP
stack[sp+4] := 0x3                // PUSH 0x3
var(1) := stack[sp+4]             // SWAP 0x1
stack[sp+4] := stack[sp+3]
stack[sp+3] := var(1)
var(2) := stack[sp+4]             // DIV
var(3) := stack[sp+3]
stack[sp+3] := (var(2) / var(3))
```

```
stack[sp+4] := 0x5              // PUSH 0x5
var(4) := stack[sp+4]           // ADD
var(5) := stack[sp+3]
stack[sp+3] := (var(4) + var(5))
var(6) := stack[sp+3]           // SWAP 0x3
stack[sp+3] := stack[sp]
stack[sp] := var(6)
var(7) := stack[sp+3]           // SWAP 0x2
stack[sp+3] := stack[sp+1]
stack[sp+1] := var(7)
var(8) := stack[sp+3]           // POP
var(9) := stack[sp+2]           // POP
var(10) := stack[sp+1]          // JUMP
jump var(10)
```

After conversion to the intermediate representation, the program leaves the front end and enters the middle end. The middle end applies several different analyses until reaching a fixed point. This includes data-flow analyses, which improve the readability of the intermediate representation.

Data-flow analyses

The purpose of the middle end is to raise the abstraction level of the intermediate representation until readable high-level code can be generated from it. The middle end accomplishes this by applying various analyses, including data-flow analyses, control-flow analyses, and function identification.

The data-flow analyses can be grouped into expression propagation and dead-code elimination. Expression propagation involves propagating the definition of a variable to its point of use when doing so is safe. For example, consider the following statements from the basic block above:

```
1 stack[sp+4] := 0x5              // PUSH 0x5
2 var(4) := stack[sp+4]           // ADD
3 var(5) := stack[sp+3]
4 stack[sp+3] := (var(4) + var(5))
```

The assignment in line 1 can be propagated into line 2, so that we get:

```
1 stack[sp+4] := 0x5
2 var(4) := 0x5
3 var(5) := stack[sp+3]
4 stack[sp+3] := (var(4) + var(5))
```

Likewise, the assignments in lines 2 and 3 can be propagated into line 4:

```
1 stack[sp+4] := 0x5
2 var(4) := 0x5
3 var(5) := stack[sp+3]
4 stack[sp+3] := (0x5 + stack[sp+3])
```

Dead-code elimination involves eliminating assignments to variables that are never used. Such unused assignments mainly arise as a result of expression propagation. For example, after the propagation above the variables defined in lines 1, 2, and 3 are never used. Elimination thus yields:

```
stack[sp+3] := (0x5 + stack[sp+3])
```

Thus the combination of propagation and elimination has raised the compactness of the code, increasing its readability.

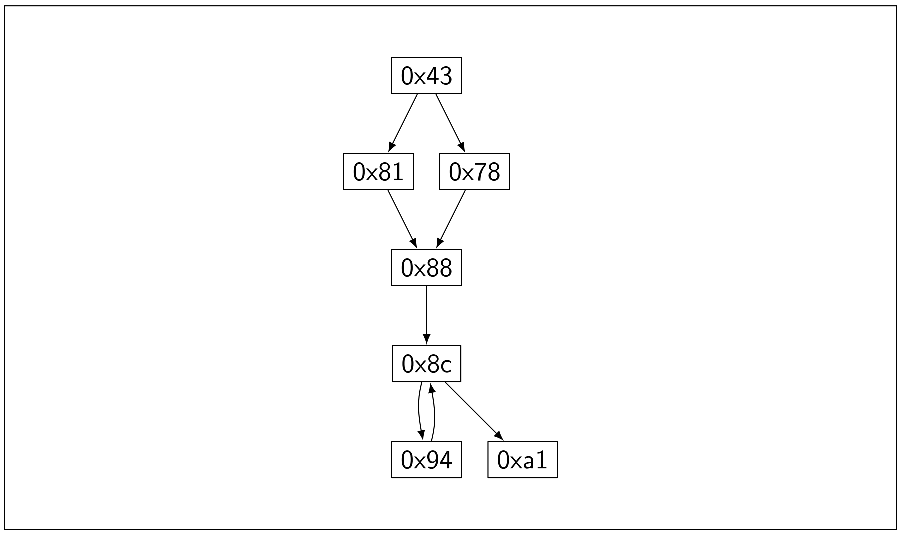
As another example, the full basic block at address 0xa1 only contains the following statements once data-flow analysis has run to a fixed point:

```
stack[sp] := (0x5 + (stack[sp] / 0x3))
jump 0x56
```

The code is clearly far more compact than immediately after conversion.

Function identification

Function identification detects that an external function exists starting with the basic block at address `0x43`. Function identification then separates out the basic blocks reachable from address `0x43` from the loader code. The CFG now more clearly corresponds to that of the original Solidity function, `myfunc` – it shows a conditional followed by a loop, and then a return:



Stack flattening

A number of other analyses run as part of the fixed-point computation in the middle end. One of these is stack flattening. Stack flattening replaces each stack variable in the `myfunc` function with a local variable, getting rid of the concepts of a stack and stack pointer.

Control-flow analyses

Once a fixed point is reached, two control-flow analyses are applied to the intermediate representation: loop structuring, and structuring of conditionals.

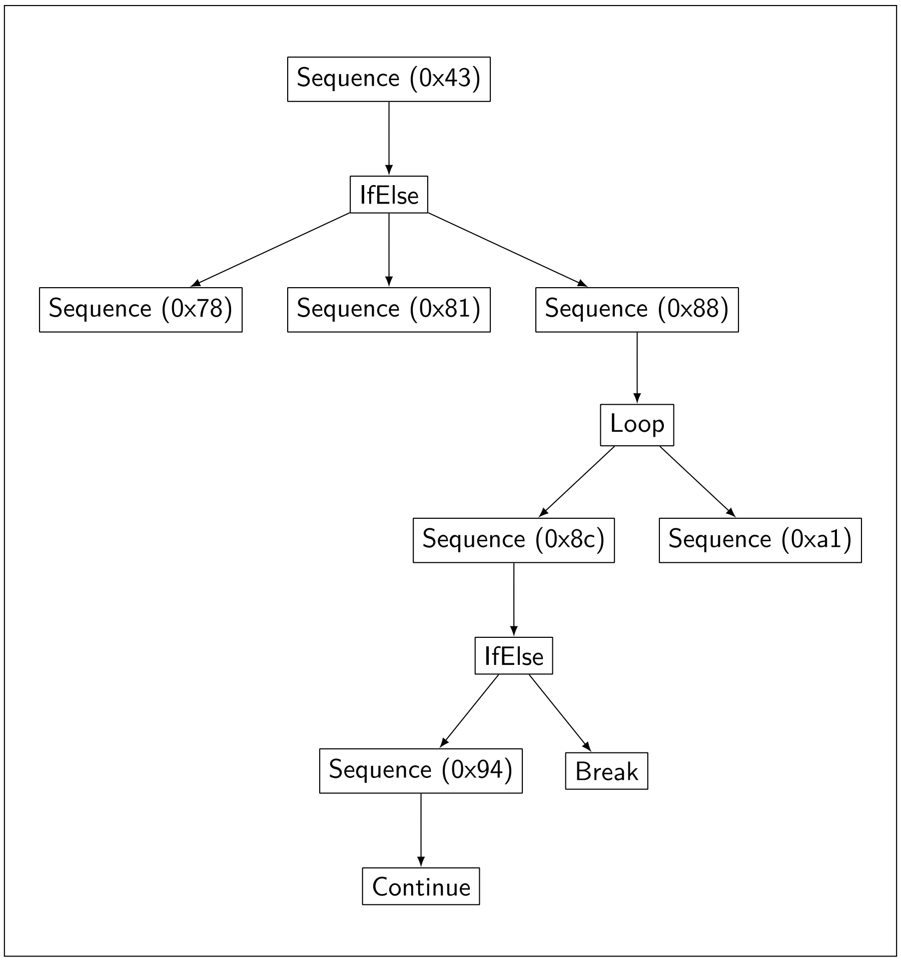
Loop structuring discovers that `0x8c` is the beginning of a loop, that `0x8c` and `0x94` are the nodes included in the loop, and that `0xa1` is the follow of the loop, meaning that control flow transfers there as the result of any `break` statements in the loop.

Structuring of conditionals involves finding the point where the two branches of a conditional meet. This makes it possible to generate more readable code in the back end. In our example, `0x43` is a conditional header, and `0x78` and `0x81` are the branches. Structuring of conditionals determines that `0x88` is the follow of this conditional.

At this point, a transformed CFG and some structuring information has been produced for each function in the smart contract. The middle end outputs these to the back end, which converts each function to an abstract syntax tree (AST) and outputs high-level code.

AST conversion

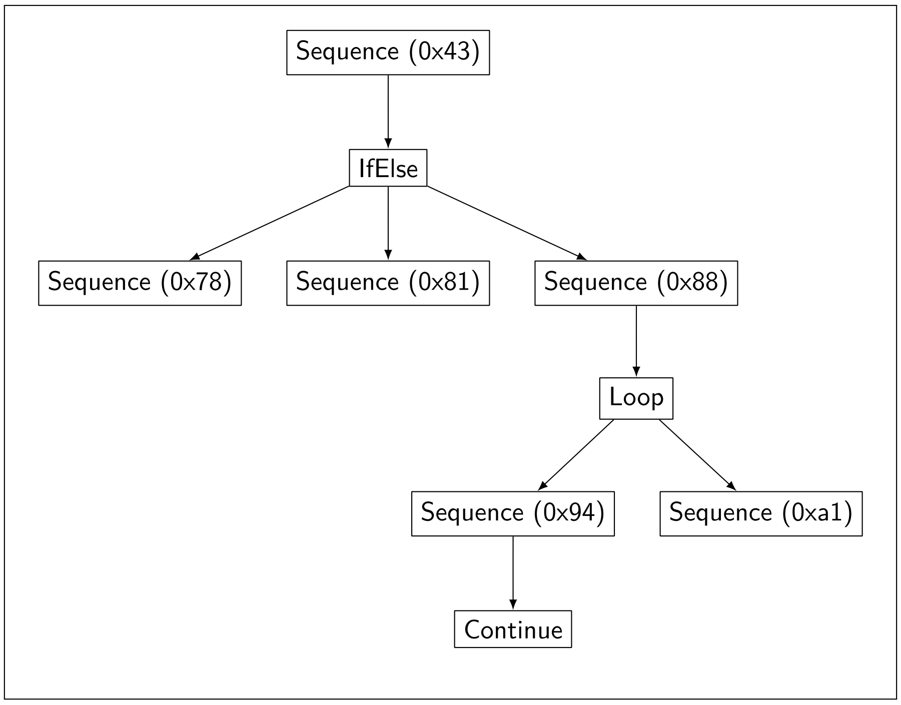
Rather than outputting high-level code directly from the intermediate representation, the decompiler first converts each function to an abstract syntax tree. The conversion happens through several passes over the code. AST node types include a `Sequence` of statements, a `Loop`, an `IfElse` node, as well as `Continue` and `Break` nodes. After conversion to an AST, `myfunc` looks as follows:



Readability improvements

After conversion to an AST, each function is processed in a readability module which makes minor semantics-preserving changes to the AST that result in more readable output.

In our example, the `Sequence` node at `0x8c` is empty and is therefore removed. The `Loop` node was initially typed as an endless loop, but now it has its condition set, becoming a pre-tested loop since its header node is an `IfElse` node that leads to a `Break`. Then loop induction variables are detected and named. The function parameter is also named. The AST now looks as follows:



Code generation

Once each function is converted to an AST, code generation is relatively simple; it involves a pass over the tree, outputting the appropriate high-level code for each AST node. For example, when visiting a `Sequence` node, code for each instruction is emitted in turn. When visiting an `IfElse` node, the initial "`if (exp) {`" is generated, then one branch is visited, then the "`} else {`" is generated, the other branch is visited, and the final "`}`" is generated.

After code generation, the final output of the decompiler is:

```
contract Decompiled {
    function loader () {
        mem[0x40:+0x20] = 0x60;
        assert(!((msg.data.length < 0x4)));
        var0 = calldataload(0x0);
        var0 = ((var0 / 0x10000...000000) & 0xffffffff);
        if ((var0 == 0xacc9d5d6)) {
            func0(calldataload(0x4));
        }
        revert(0x0, 0x0);
    }

    function func0 (param0) {
        assert(!(msg.value));
        if ((0x1 == (param0 % 0x2))) {
            var0 = 0x1;
        }
        else {
            var0 = 0x2;
        }
        i = 0x0;
        while ((i < param0)) {
            var0 = (0x2 + var0);
            i = (0x1 + i);
            continue;
        }
        haltreturn((0x5 + (var0 / 0x3)));
    }
}
```

With that, the `SmallExample` contract is successfully decompiled. We have now provided an overview and intuition of each phase in our decompiler. The following chapters describe each phase in more detail.

5 FRONT END

The decompiler takes EVM bytecode as its input. The purpose of the front end is to convert this bytecode to an intermediate representation that the middle end can analyze. The front end consists of two phases, parsing and conversion. In the parsing phase, the bytecode is split into EVM instructions and their associated operands. The instructions are divided into basic blocks. During conversion, each EVM instruction is turned into a sequence of statements in our chosen intermediate representation.

5.1 PARSING

The parsing phase turns EVM bytecode into basic blocks of EVM instructions. Because the EVM is stack-based, operands for almost all instructions are implicit. As such, EVM *byte*code lives up to its name, using up exactly one byte per instruction. This makes parsing straightforward. The only exception is the set of `PUSH` instructions, `PUSH1`, `PUSH2`, ..., `PUSH32`, which are followed by a number of bytes forming the operand to the instruction. A comprehensive mapping from EVM opcodes to instructions can be found in the formal definition of the Ethereum protocol [23].

The 32 different `PUSH` instructions have different opcodes. For simplicity we group these into a single instruction with an operand during parsing. The same applies to the `DUP` and `SWAP` instructions. For example, the `DUP3` opcode becomes a `DUP` instruction with an operand of 3.

Instructions are split into basic blocks during parsing. Whenever an instruction that transfers control is encountered, the current basic block ends and a new one begins. Examples are the `JUMP` and `JUMPI` instructions. The current basic block also ends if an instruction causes execution to halt.

We must ensure that no jumps can target the middle of a basic block. This is accomplished by ending the current basic block and starting a new one when the `JUMPDEST` instruction is encountered. (The EVM ensures that jumps can only target such instructions, otherwise the transaction is reverted.)

After parsing, the low-level instructions in each basic block need to be converted into an intermediate representation to ease analysis in the middle end.

5.2 INTERMEDIATE REPRESENTATION

In our chosen intermediate representation, each function in the program is represented as a control-flow graph (CFG), consisting of a number of basic blocks connected through edges representing possible flows of control.

A basic block contains a list of instructions, where the last instruction is always one that transfers control flow to another basic block or terminates the program. The instructions have types, such as `assign` and `jump`. Instructions operate on expressions, which we describe momentarily. Statements can have side effects, while expressions cannot. The instructions in our intermediate representation are described in Figure 5.1.

Most instruction types are self-explanatory. Worth noting is the `Assertion` instruction that reverts the transaction if its condition evaluates to zero.

```
Instruction ::= Assign(Variable, Exp)
              | Jump(Loc)
              | JCond(Loc, Exp)
              | VMCall(Name, Args, Results)
              | Call(Loc, Args, Results)
              | Ret(Args)
              | Assertion(Exp)
Args        ::= Exp*
Results     ::= Exp*
Loc         ::= Exp
Name        ::= String
```

Figure 5.1: A description of the instructions in the intermediate representation

Another special instruction is `VMCall`. It is used to represent functionality that exists only in EVM, not in Solidity. For instance, EVM has the `MSIZE` instruction that computes the size of used memory. Solidity does not have this low-level concept, so we represent it through a VMCall with `msize` as the called functionality and hope to get rid of the instruction during processing in the middle end.

5.3 EXPRESSIONS AND VARIABLES

Instructions operate on expressions; these can be literals, unary- or binary operators, or variables. Figure 5.2 lists the different types of expressions.

```
Exp         ::= Literal | UnaryOp Exp | Exp BinaryOp Exp
              | Sequence(Exp*) | Variable
UnaryOp     ::= not | neg
BinaryOp    ::= add | sub | mul | div | mod | xor | gt
              | lt | and | or | equal | exponent | mod
Variable    ::= LocalVar
              | GlobalVar(Name)
              | Stack(Literal)
              | Mem(Exp, Exp)
              | Storage(Exp, Exp)
```

Figure 5.2: A description of the expressions in the intermediate representation

It is worth noting that some variable types, such as memory- and storage variables, have a base address and a length, both of which are generic expressions rather than literals. This means that these variable types can get complex. For example, `Storage(Mem(var0, var1), var1)` is valid, and such expressions are not uncommon when decompiling realistic contracts.

As a consequence, it is not immediately clear whether two variables are aliases of one another, i.e., refer to the same location. For example, one cannot immediately tell whether `Mem(var0, 0x20)` and `Mem(var1, 0x20)` refer to the same location without further analysis. This problem frequently crops up for stack variables: given two stack variables, such as `stack[sp+3]` and `stack[sp-1]`, they may or may not refer to the same location depending on how the `sp` variable changed between their respective program points.

Determining whether variables potentially *may* refer to the same location, or if they definitely *must* refer to the same location is known as alias analysis

[29]. Our compiler needs at least basic alias analysis in the middle end. We describe our implementation of a simplistic alias analysis in Section 11.5.

5.4 conversion

The conversion module converts basic blocks of EVM instructions into basic blocks of IR statements.

All in all, the conversion module converts one basic block at a time. Converting a basic block involves converting each instruction in it.

Since the EVM is stack-based, instructions take operands from the program stack. Our intermediate representation therefore includes the concept of a stack variable, facilitating the conversion of such instructions.

Due to our design of the intermediate representation, many EVM instructions can be trivially converted. For example, the `JUMP` instruction is converted into `sp = sp - 1; Jump(stack[sp+1])`. The `Assign` IR statements lets us trivially represent `DUP`, `SWAP`, `PUSH`, etc.

Arithmetic instructions are converted in a straightforward manner. For example, the `ADD` EVM instruction is converted into the following sequence of IR assignments:

```
stack[sp-1] = stack[sp] + stack[sp-1];
sp = sp - 1;
```

where `sp` is considered a global variable.

Some EVM instructions have a direct counterpart in Solidity. For example, the `CALLER` instruction pushes the address of the account that initialized the current method call. This information is available through the `msg.sender` global variable in Solidity. Conversion of such instructions is therefore trivial; for example, the `CALLER` instruction becomes:

```
sp = sp + 1;
stack[sp] = GlobalVar("msg.sender");
```

Similarly, some instructions correspond directly to a Solidity built-in function; e.g., the `REVERT` instruction corresponds to the `revert()` Solidity function. We represent such instructions by generating a `VMCall` statement. We do not use the `Call` statement because then we can distinguish between calls to user code and calls to built-in functions.

In contrast, certain instructions have no counterpart in Solidity. A trivial example is the `PC` instruction, which retrieves the value of the program counter. This concept exists only in EVM, not in Solidity. We represent such EVM instructions using the `VMCall` IR statement. We think of this as a call to a built-in `pc` function. Such vmcalls are ideally removed before code generation; otherwise they can be represented using inline EVM assembly.

All of the EVM instructions can be converted using the concepts we have presented so far. Figure 5.3 provides more examples of how instructions are converted. The semantics of each instruction, as well as its operands and results, can be found in the Ethereum yellow paper [23].

All changes to the `sp` variable are consolidated on a per-block basis; the total `sp`-change is kept in each basic block. As an example, consider the following basic block, which computes `1+2`:

```
sp = sp + 1;
stack[sp] = 1;
sp = sp + 1;
stack[sp] = 2;
stack[sp-1] = stack[sp-1] + stack[sp];
sp = sp - 1;
```

EVM instruction	IR statements)
`PUSH 123`	`sp = sp + 1;`
	`stack[sp] = 123;`
`POP`	`sp = sp - 1;`
`DUP1`	`sp = sp + 1;`
	`stack[sp] = stack[sp-1];`
`SWAP1`	`stack[sp+1] = stack[sp];`
	`stack[sp] = stack[sp-1];`
	`stack[sp-1] = stack[sp+1];`
`STOP`	`VMCall("stop", [], []);`
`JUMP`	`sp = sp - 1;`
	`Jump(stack[sp+1]);`
`JUMPI`	`sp = sp - 2;`
	`JCond(stack[sp+2], stack[sp+1]);`
`ADD`	`stack[sp-1] = stack[sp] + stack[sp-1];`
	`sp = sp - 1;`
`LT`	`stack[sp-1] = stack[sp] < stack[sp-1];`
	`sp = sp - 1;`
`SHA3`	`sp = sp - 1;`
	`VMCall("sha3",`
	` [Mem(stack[sp+1], stack[sp])], [stack[sp]]);`
`ORIGIN`	`sp = sp + 1;`
	`stack[sp] = GlobalVar("tx.origin");`

Figure 5.3: Examples of how various EVM instructions are converted to the intermediate representation

After consolidation, the basic block becomes:

```
sp = sp + 1;
stack[sp] = 1;
stack[sp+1] = 2;
stack[sp] = stack[sp] + stack[sp+1];
```

This step makes analysis within a basic block far simpler, because it is immediately obvious whether two stack variables are equal.

To produce valid Solidity code, all stack variables must be removed, since Solidity does not have the explicit concept of a stack. This can be accomplished through stack flattening, which turns each stack location into a local variable. This is not the responsibility of the front end, however; our experience shows that too much imprecision remains at this stage for stack flattening to be feasible. Stack flattening occurs in the middle end once the precision of the CFG has been increased sufficiently.

Once the program is converted to the intermediate representation, it is passed on to the middle end, where various analyses are iterated to a fixed point. The details of these analyses are described in the following chapters.

6 DATA-FLOW ANALYSIS

Data-flow analysis involves expression propagation and dead-code elimination. These analyses are applied to increase the abstraction level, and thus readability, of the decompiler output.

For example, consider the following statements, which could be the result of a direct conversion from EVM bytecode to our intermediate representation:

```
var0 = 3
var1 = 5
var2 = var0 + var1
```

These statements have a low abstraction level because they are produced by direct conversion from EVM instructions, which can only take operands from the top of the program stack.

The statement `var2 = var0 + var1` uses the variables `var0` and `var1`. Expression propagation therefore attempts to locate possible definitions of these variables. These variables are defined by the statements `var0 = 3` and `var1 = 5`, respectively. The right-hand side of the definitions are then propagated to the point of use, resulting in the following code:

```
var0 = 3
var1 = 5
var2 = 3 + 5
```

At this point, neither `var0` nor `var1` are used. Dead-code elimination therefore can safely eliminate the assignments. We are left with the code:

```
var2 = 3 + 5
```

This code is of a higher abstraction level than the original code. It is more compact, and arguably more readable.

6.1 EXPRESSION PROPAGATION

The goal of expression propagation is to propagate the definition of a variable to the point of use of said variable whenever it is safe to do so.

Expression propagation recovers more complex expressions from simpler ones, increasing the abstraction level of the code. This is possible because our IR supports arbitrarily complex expressions, unlike EVM bytecode.

Expression propagation has the additional benefit of resolving indirect jump targets, increasing the precision of our CFG, and thus improving the performance and accuracy of every analysis.

Expression propagation is applied to every variable in the program, one at a time. This involves two phases. First, the possible definitions of the variable must be discovered. The definition must be unique, otherwise propagation is not safe. Then it must be ascertained that none of the variables appearing on the right-hand side of the definition are redefined before the use-point is reached.

For example, consider the following statements:

```
var0 = 3
var1 = 5
var2 = var0 + var1
var1 = var1 + 1
var3 = var2
```

To propagate the use of `var2` in the statement `var3 = var2`, the unique definition `var2 = var0 + var1` is found. However, since `var1` is redefined between the definition and use of `var2`, propagation is *not* safe in this example. If we did proceed to propagate anyway, we would get:

```
var0 = 3
var1 = 5
var2 = var0 + var1
var1 = var1 + 1
var3 = var0 + var1
```

which is unsound – now `var3` ends up holding the value `9` instead of `8`, as it would before propagation.

In other words, for expression propagation to be safe, these two conditions must hold [3]:

– The definition must be unique.
– Each variable appearing in the right-hand side of the definition must not be redefined along any path from the definition to the point of use.

Since the second condition talks about *any* path, we need an all-paths analysis to determine whether expression propagation is safe.

In the example we have given, each variable is a local variable; it is trivial to check whether two local variables are the same. Testing variable equality is significantly harder when variables can be indirect, which is the case for memory- and storage variables. Similarly, reasoning about whether two stack variables in different basic blocks are the same requires reasoning about the value of the `sp` variable. This introduces imprecision while computing the possible definitions of some variable types. It also means that the all-paths analysis becomes less precise. Both have to make use of a *may*-analysis to check if two variables may be the same.

The algorithm for expression propagation is given as pseudocode in Figure 6.1. The algorithm uses two helper functions, `find_possible_definitions` and `is_redefined_along_path`. Both of these helpers must perform the variable equality checks we just described, using a *may*-analysis. They therefore need to keep track of changes to the `sp` variable across basic blocks.

6.2 DEAD-CODE ELIMINATION

Dead-code elimination involves eliminating any instructions that are safe to eliminate. We refer to such unnecessary statements as "dead code". While modern optimizing compilers may not produce much dead code, expression propagation results in many statements becoming dead. In the example code from above:

```
var0 = 3
var1 = 5
var2 = 3 + 5
```

The assignments to `var0` and `var1` have become dead code after expression propagation. They can therefore be eliminated.

Dead-code elimination is an important analysis, because without it, the generated code would be riddled with unnecessary assignments left over by expression propagation, and would therefore be nearly unreadable.

It is safe to eliminate an assignment instruction when the defined variable is never used in the future. This means that to perform dead-code elimination, an all-paths analysis is needed starting at the definition. As with expression

```
function propagate_expressions (f) {
  for each basicblock in f {
    for each instruction in basicblock {

      // attempt to propagate each used variable
      for each used_var in used_vars(instruction) {

        // find possible definitions
        use_point = (basicblock, instruction, used_var);
        defs = find_possible_definitions(use_point);

        // definition must be unique
        if (length(defs) != 1) {
          continue;
        }

        def_point = defs[0];
        def_ins = def_point.instruction
        rhs_vars = used_vars(def_ins);

        // check that no variable from the definition is
        // redefined along any path from definition to use
        safe_to_propagate = true;
        for each path in all_paths_between(def_point,
                                           use_point) {
          for each rhs_var in rhs_vars {
            if (is_redefined_along_path(rhs_var, path)) {
              safe_to_propagate = false;
            }
          }
        }

        if (safe_to_propagate) {
          instruction.replace_uses(used_var, def_point.rhs);
        }
      }
    }
  }
}
```

Figure 6.1: The algorithm for expression propagation

propagation, variable equality can only be approximated, and the analysis can gain precision by keeping track of sp variable changes.

7 FUNCTION IDENTIFICATION

A Solidity function can be marked as `external` or `internal`.[1] An `external` function can be invoked as the result of a transaction, while an `internal` function can only be invoked internally, i.e., from within the contract.

Recall that initially, we consider all of the contract code as belonging to a single large function, namely the contract loader code, which starts at address `0x0`. Function identification aims to identify and separate out the basic blocks that can safely be moved to separate functions.

Function identification is a crucial part of decompilation for two reasons. First, with function identification the code is split into many smaller parts, each of which can be understood individually, improving readability [43]. Additionally, our analyses perform far better on many small functions than a single large one, since the performance of some inter-basic block analyses scales super-linearly with the number of edges in the analyzed CFG.

7.1 EXTERNAL FUNCTION IDENTIFICATION

We identify external functions separately, since they are much easier to identify than internal ones, and can be separated out using simple pattern matching on the loader code. External function identification does not require a very precise CFG nor any expensive analyses. Identifying external functions through pattern-matching on the loader code is also done by the Ethersplay [10] analysis tool for EVM bytecode, from which we got the idea.

All calls to external functions go through the loader code. The loader code uses the `calldataload` instruction to load the first four bytes of user input. These bytes identify the function to call. The loader cases over these bytes, and when a match is found, it jumps to the external function.

Figure 7.1 shows an example of the decompiled loader code after external function identification. Before the external functions are identified, the basic blocks of each external function reside in the loader code, such that the program is one large function.

An external function first uses the `calldataload` instruction to load its arguments. It then performs the actual work of the function. Finally the `return` instruction is used to halt execution and return the result of the call.

Our external function identification works through pattern matching on the loader code. It looks for the `calldataload` of the first four bytes of input. It then identifies any conditional jumps based on these four bytes after a comparison against a literal.

When such a conditional jump is identified, the destination basic block, and any basic block reachable from it, is split off into a separate function. The jump to the basic block is replaced with a `call` statement. (Note that a function call statement only exist in our intermediate representation, not in the EVM instruction set.)

Once the basic blocks of an external function have been determined through a depth-first search, parameters are identified by analyzing the use of `calldataload` instructions. Typically, the first four bytes identifying the function

[1] A contract can also be marked as `public`, which is a generalization of `external`, and as `private`, which is a specialization of `internal`. The details are not important here.

```
function loader () {
    if ((msg.data.length < 0x4)) {
        revert(0x0, 0x0);
    }
    var0 = calldataload(0x0);
    var0 = (var0 / (2 ** (256 - 32)));
    var0 = (0xffffffff & var0);
    if ((0x6482e626 == var0)) {
        func0(calldataload(0x4));
    }
    else if ((0x1c31f710 == var0)) {
        func1(calldataload(0x4), calldataload(0x24));
    }
    else if ((0x625fcce7 == var0)) {
        func2();
    }
    else if ((0xc3da42b8 == var0)) {
        func3();
    }
    else {
        revert(0x0, 0x0);
    }
}
```

Figure 7.1: The decompiled loader code of an example contract after external function identification.

are retrieved using `calldataload(0x0)`. Then the first function argument is loaded using `calldataload(0x4)`, the next using `calldataload(0x24)`, and so on. Return values can be identified as the input to the `haltreturn` vmcall that comes from the `RETURN` EVM instruction; this input is a memory address where the packed return values reside.

It may happen that two different external functions both call the same internal function. In that case there is an overlap in the basic blocks pertaining to each function. We resolve this issue by duplicating any basic blocks belonging to more than one function, which corresponds to inlining the internal functions. As the precision of the CFG is improved, internal function identification may be able to identify that these basic blocks belong to a separate internal function and split them off accordingly.

7.2 INTERNAL FUNCTION IDENTIFICATION

The aim of internal function identification is to detect and separate out the code of internal functions. Internal functions are functions that cannot be directly called as the result of a transaction; control only flows to internal functions indirectly through external function calls.

The easiest solution is to never identify internal functions at all, instead simply inlining their code in the external functions. While this produces correct code, the code can often become significantly more readable if internal functions are identified.

When an internal function is identified and separated out, it can be understood separately, increasing readability [43]. Additionally, the same code is then not inlined in multiple places; this increases the compactness of the high-level code. Also, when an internal function is not identified, its final return statement appears in the intermediate representation as an indirect jump. This obscures the control flow by producing a `goto` statement during code generation. It is therefore crucial to discover as many internal functions as possible.

A heuristics-based approach

Initially, we attempted to identify internal functions through a heuristic; a function call typically manifests itself as a push of the return address, then a push of the arguments, then a push of the function address, and then a jump instruction. We used this heuristic to detect which basic blocks are function headers. Function returns manifest as indirect jumps. We therefore assumed that all indirect jumps are returns, in order to discover where a function ended.

Unfortunately this approach did not work for smart contracts of realistic size. Sometimes the return address is placed on the stack in a separate basic block, and the pattern of pushes cannot be detected. Sometimes multiple return addresses are pushed in sequence, as a tail-call optimization by the compiler.[2] Thus the heuristic would often fail, and it would sometimes create a function when there was none, resulting in an incorrect decompilation. We therefore needed a more principled approach.

Effects of function creation

Our current approach is to only separate code into a new internal function when it is definitely safe to do so. Essentially, internal function identification works by considered each basic block *h* in the program in turn, and checking if *h* can safely be turned into the header node of a new function *f*.

The question, then, is which conditions must be checked before a function can safely be identified and separated out. To answer this question, we considered which changes to the program occur when a function is identified. It must be possible to make all these changes without altering the semantics of the program.

When an internal function *f* with header node *h* is identified, the following changes occur:

1. Each predecessor *p* of *h* has its jump to *h* replaced with a `call` statement.
2. For each predecessor *p*, the return address is analyzed to determine to which address control-flow is transferred after the call, and a `jump` statement is inserted into *p* after the `call` statement.
3. The basic blocks of *f* are moved out of the current function and into their own, separate function.
4. The indirect jumps inside *f* that have the return address as destination are converted into `return` statements.
5. The number of parameters and return values of *f* are determined, and then:
   (a) The arguments for each call to *f* are added to the relevant `call` statement according to the discovered number of parameters.
   (b) The return values for each call to *f* are added to the relevant `return` statements according to the discovered number of return values.

[2] For example, for the call `f(g())` the compiled EVM instructions could push the return address, then the address of `f`, then jump to g; when g returns, it jump to `f`, which computes its value and returns to the return address.

Determining the basic blocks of a function

Assuming that it is safe and feasible to make the changes just described, the basic blocks of *f* must be determined and moved into a separate function.

To determine which basic blocks belong to *f*, we define the *reach* of a basic block *n* recursively as:

$$\text{reach}(n) = \begin{cases} \{n\}, & \text{if } n \text{ is a return} \\ \{n\} \cup \bigcup_{s \in \text{successors}(n)} \text{reach}(s), & \text{otherwise} \end{cases}$$

In plain terms, the reach of *h* includes any basic block reachable from *h* without following the successors of any return statements. Thus the basic blocks included in a function *f* with header node *h* is simply the set reach(h).

Safety requirements

To safely perform the changes described above, a number of conditions must hold.

For change (1) to be safe, every predecessor of *h* must reach *h* through a direct jump. For change (2) to be safe, it must be known with certainty what the value of the return address is for every predecessor of *p*; this value is placed onto the stack before the function call. Note that this is *not* always the address of the next basic block, due to the tail-call optimization described earlier. Change (3) is always safe. For change (4) to be safe, it must be known which of the indirect jumps in *f* jump to the return address. For change (5) to be safe, we must be able to determine the number of parameters and return values of the function. The process is described in more detail in Section 7.2, but it requires the height of the stack to be known at any basic block inside the function.

In addition to the safety requirements above, there is another requirement which should hold before we are ready to think of *f* as a conventional function. We require that that there be no jumps "into the middle of *f*"; that is, for each node $n \in (\text{reach}(h) - \{h\})$ it must hold that:

$$\text{predecessors}(n) \subseteq \text{reach}(h)$$

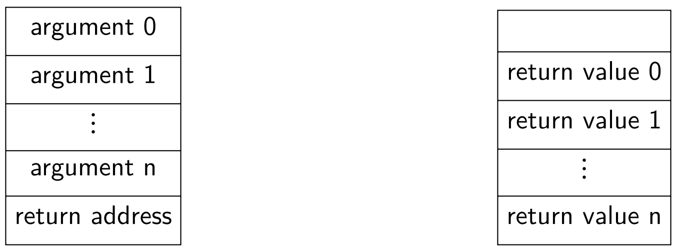
Additionally, reach(h) should never include the header of another function.

If all these safety requirements are met for some basic block *h*, then internal function identification can safely turn reach(h) into a new function, performing changes (1) - (5) described above.

Counting parameters and return values

When a function is called, the EVM calling convention dictates that the top word of the stack holds the first argument, the second word holds the second argument, and so on. The return address resides immediately after the last argument. The stack layout at call-time is illustrated in Figure 7.2a. After a function has performed its work, the return address is removed from the stack, and the return values are placed on top of the stack. This is illustrated in Figure 7.2b.

The overall strategy to determine the number of parameters and return values of a function *f* is as follows. We compute the stack depth at which the return address resides. Each word that resides above the return address at call-time is a function parameter. Thus, knowing the offset of the return



(a) The stack layout before a call (b) The stack layout after a call

Figure 7.2: Stack layouts

address lets us compute the number of function parameters. Once we know this, we can compute how much the stack height changes from the function entry until a `return` statement is reached. This delta lets us compute the number of return values.

As a concrete example, assume that at call-time the return address of *f* resides at `stack[sp-3]`, and function execution changes the stack-height by -2. Then we know that the values above the return address, namely `stack[sp]`, `stack[sp-1]` and `stack[sp-2]`, are the function arguments. Thus *f* has 3 parameters. We also know that *f* must remove its parameters and return address from the stack during its execution, and the number of return values is the number of words that remain in the stack frame afterwards. Since the stack delta is -2, then the number of return values is $3 + 1 - 2 = 2$.

To discover the offset of the return address, we make the assumption that a function never accesses any part of the stack below its stack frame. That is, the deepest stack-value accessed by a function is the return address, since the arguments reside above it. Thus to discover where on the stack that the return address resides, our analysis determines the deepest `sp`-offset used anywhere in the reach of the discovered function.

Since the `sp` variable typically changes from one basic block to another, it is necessary to determine the height of the stack at each basic block within the function. If the stack height cannot be uniquely determined, the function cannot be safely identified.

In short, to determine the number of parameters and return values, our analysis finds the deepest `sp`-offset, assumes the return address resides there, uses this information to compute the number of parameters, and uses how much the `sp`-value changes throughout the function to compute the number of return values.

We have now explained how external and internal functions are identified: external functions are found through a heuristic, while internal functions are discovered by checking if each basic block satisfies the safety conditions described, and making it a function header node if so. Besides function identification and the data-flow analyses, the middle end applies a number of other analyses which are described in the following chapter.

8 OTHER ANALYSES

Our decompiler design includes various analyses that simplify the representation of the program, either to improve readability or to increase the effectiveness of other analyses. Examples of such analyses are reconstructing `assert` statements, merging of basic blocks, and removing stack variables through stack flattening. This chapter describes these miscellaneous analyses.

8.1 ASSERT RECONSTRUCTION

The intermediate representation sometimes contains a conditional jump to a basic block that reverts the current transaction. Such conditional jumps are relatively common; they naturally arise when there is an `assert(condition);` statement in the original Solidity source code, as there is no `assert` EVM instruction. However conditional jumps to reverts may also be inserted by the compiler for various other reasons, e.g., to check that an array access does not go out of bounds, or to ensure no currency is transferred unless the called function is marked as `payable`.

Such conditional jumps can be replaced with an equivalent `assert` statement. Performing such a transformation is useful because it simplifies the control-flow graph that is being analyzed, resulting in better precision and performance of other analyses. It also means that control-flow analysis has to identify fewer conditionals.

Our analysis looks for any conditional jumps to a basic block that only has a single instruction, invoking the `revert` vmcall. When such conditional jumps are identified, they are replaced with an `assert` instruction, asserting the negation of the expression in the conditional jump. The basic block terminator becomes an unconditional jump to the following basic block. Similarly, our analysis looks for conditional jumps falling through to a basic block that invokes `revert`; in case of such fall-throughs the condition need not be negated.

An example of a conditional jump to a `revert` vmcall is shown in Figure 8.1.

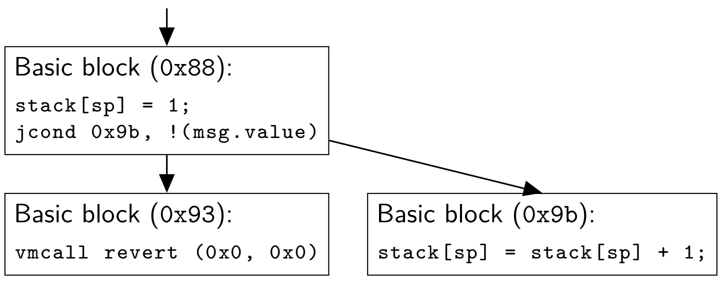

Basic block (0x88):
```
stack[sp] = 1;
jcond 0x9b, !(msg.value)
```
Basic block (0x93):
```
vmcall revert (0x0, 0x0)
```
Basic block (0x9b):
```
stack[sp] = stack[sp] + 1;
```

Figure 8.1: An example of a conditional jump to a `revert` statement

After the transformation is applied, the resulting basic blocks are shown in Figure 8.2.

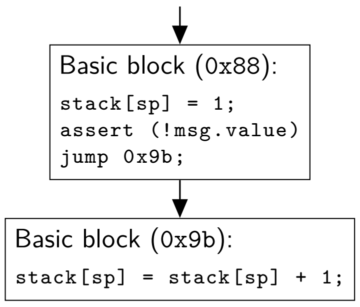


Figure 8.2: The basic blocks of Figure 8.1 after assert reconstruction

8.2 MERGING OF BASIC BLOCKS

Sometimes, a basic block B_1 has a single successor B_2, and B_2 has B_1 as its single predecessor. In that case it is safe to merge the two basic blocks.

Merging of basic blocks is desirable because it improves the precision of the decompiler; it is easier to be precise in an intra-BB analysis compared to an inter-BB one, e.g., because reasoning about the stack pointer is trivial within a basic block. Merging is also desirable for performance reasons, as having fewer blocks to process results in a speedup. Additionally intra-BB propagation and elimination are far cheaper than inter-BB ones, since they do not have to perform an all-paths analysis. Thus the more work can be performed in the intra-BB analyses instead of the inter-BB ones, the faster decompilation terminates.

Merging of basic blocks B_1 and B_2 is safe provided that the following conditions hold:

– B_2 is the sole successor of B_1.
– B_1 is the sole predecessor of B_2.
– B_1 is not a function header.

To merge B_1 and B_2 into a new basic block B', the instructions of B_1 and B_2 must be concatenated to form the instructions of B'. Recall that changes to the global stack pointer variable are kept track of on a per-basic block basis. Thus before concatenation, the instructions of B_1 which use a stack variable must be adjusted to account for the change of the stack pointer in B_2. The new basic block B' has the predecessors of B_1, the successors of B_2, and the address of B_1.

As an example, consider the basic blocks in Figure 8.3. Since the two basic blocks satisfy the previously mentioned conditions, they can safely be merged.

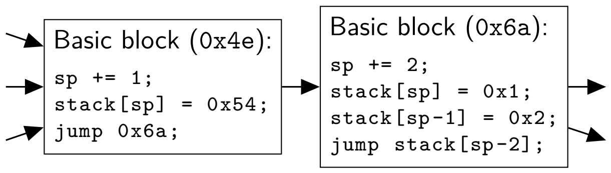
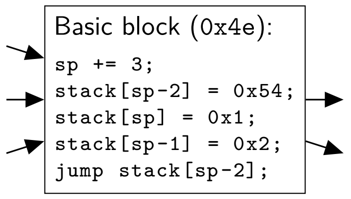


Figure 8.3: An example of two basic blocks that can be safely merged

The merging process involves adjusting the stack variable `stack[sp]` in the first basic block according to the sp-delta of the second basic block, which is +2. The variable therefore becomes `stack[sp-2]` after adjustment. The final merged basic block is shown in figure 8.4.



```
sp += 3;
stack[sp-2] = 0x54;
stack[sp] = 0x1;
stack[sp-1] = 0x2;
jump stack[sp-2];
```

Figure 8.4: The basic blocks of Figure 8.3 after merging

After merging, a cheap and simple intra-basic block expression propagation would suffice in order to determine that the jump destination is `0x54`, making it possible to reduce the number of successors of the basic block to one, potentially enabling further merging.

8.3 STACK FLATTENING

Stack flattening is an analysis that turns each stack variable in a function into a local variable; after flattening a function, the code in the function no longer has references to any stack variables, nor to the global stack pointer variable, `sp`.

It is highly desirable to perform stack flattening, because while the Ethereum VM does feature a stack, high-level Solidity code has no such concept. Thus stack flattening results in higher-level code that more closely resembles Solidity. Additionally, it is far easier for other analyses to reason about local variables than stack variables, since stack accesses are always indirect through the `sp` variable, which introduces imprecision.

Stack flattening is a concept that also arises in the decompilation and analysis of Java bytecode [48, 45]. This is unsurprising, since both the EVM and the Java VM are stack-based virtual machines. However stack flattening is simpler for Java bytecode, because the JVM specification guarantees that each program point has a unique stack height [48].

The general idea in stack flattening is to turn every stack slot into its own variable. Thus we can replace all occurrences of `stack[sp+0]` with `var0`, all occurrences of `stack[sp-1]` with `var1`, and so on.

In order to safely perform stack flattening, our analysis must compute the stack height at the beginning of each basic block. If the stack may have more than one possible height at any program point, stack flattening can not be done safely, since the stack slots no longer correspond to local variables in a one-to-one manner. As a consequence, whenever the analyzed CFG has imprecision, e.g., in the form of indirect jumps, stack flattening typically becomes impossible for the given function. The imprecision of indirect jumps introduces new paths in the CFG, and these paths often disagree with other paths on the stack height at some program point.

Provided that the CFG for a function is sufficiently precise and the stack height can be uniquely determined at each basic block, stack flattening is applied. Recall that in our intermediate representation, changes to the `sp` variable are consolidated on a per-basic block basis. This means that each basic block has an `sp` delta.

Our analysis works by first traversing the CFG, keeping track of the stack height at each basic block by analyzing the `sp` delta values. Once the stack height is known, the `sp` delta of each basic block is adjusted to zero, which involves modifying the offset of each stack variable in the basic block. Finally a mapping is set up from stack variables to local variables, and all occurrences of stack variables are replaced with the corresponding local variable.

8.4 successor reduction

The successor reduction analysis is intended to refine the over-approximation of possible successors for any indirect jumps in the program.

Any basic block that ends in an indirect jump has a set of possible successors, which is a sound over-approximation. We initially use the set of all possible basic blocks as a trivial over-approximation to this set.

Decreasing the size of this set makes every other analysis significantly faster and more effective. The inter-basic block expression propagation analysis benefits especially much, since it involves an all-paths analysis on the CFG. Better expression propagation in turn results in an even more precise CFG. Successor reduction is therefore a crucial analysis.

In EVM code, jump targets are pushed to the program stack at one point during the program. A later `JUMP` instruction pops this word off the stack and transfers control to it. We assume that every jump target in the program is placed on the stack as the result of a `PUSH` instruction. In other words, we assume that all jump targets are literals rather than being the result of a complex computation. [1]

This assumption allows us to narrow down the possible targets of an indirect jump by removing those successors whose address does not appear as a literal anywhere in the program.

As iterated dead-code elimination removes literals from the program, it is useful to reapply the successor reduction afterwards. Thus this analysis should be iterated to a fixed point along with the other analyses in the middle end.

8.5 constant folding

Our decompiler attempts to reduce expressions as much as possible through constant folding. Arithmetic identities such as `var0 - var0 == 0` and `var0 / 1 == var0` are used to simplify expressions further. This is occasionally useful to determine at decompile-time whether a conditional jump is taken or not.

[1] This assumption has held for every program we have inspected so far. However it is easy to imagine that such jump destinations would be prime targets for obfuscation of EVM code.

9 CONTROL-FLOW ANALYSIS

The control-flow analysis (CFA) phase of the decompiler discovers and structures loops. The result is a more readable high-level output; if a loop cannot be structured, `goto` statements are output during code generation. This phase also structures conditionals, which means discovering the point where the two branches meet. This results in code with a lower nesting level and fewer `goto` statements.

9.1 STRUCTURING LOOPS

Various approaches to loop structuring exist, and the area has been the target of recent research. For simplicity we decided to use Cifuentes's algorithm, which is based on interval analysis. This section is therefore heavily based on Cifuentes' PhD thesis [3].

Structuring a loop involves discovering that a cycle exists; determining which nodes to include in the loop; determining the loop type (pre-tested, post-tested, or endless); and computing the follow node of the loop [3]. We first introduce the necessary interval theory, and then we explain how each of these steps is accomplished.

Interval theory

The loop structuring algorithm we use is based on the graph-theoretic concept of an *interval*. Finding the intervals in a CFG allows the discovery of loops, because each interval corresponds to at most one loop, and intervals provide a nesting order for loops. Intervals were first defined by J. Cocke [34] as a graph-theoretic concept, which was found to be useful in control-flow analysis in compilers; see Allen [30] for example applications.

An interval is defined in terms of a node in a graph. So let $G = (V, E)$ be a graph consisting of a set of nodes V and a set of edges E. Let $h \in V$ be a node in V. Then we define $I(h)$ (the *interval* of h) as the maximal subgraph for which it holds that h is the only entry node, and every cycle goes through h. We refer to h as the *header node* of the interval.

A graph can be partitioned into a unique set of disjoint intervals [3]. A concrete example of a graph partitioned into intervals is given in Figure 9.1. Nodes 1 and 2 are header nodes. Intervals are marked with dotted boxes. $I(1)$ does not correspond to a loop, but $I(2)$ does due to the back-edge $(3, 2)$. It is also worth noting that node 4 belongs to $I(2)$ despite not being part of the loop nodes.

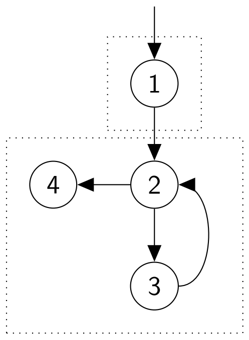


Figure 9.1: An example of a graph partitioned into intervals

An algorithm for partitioning a graph into intervals is given in Figure 9.2, which is adapted from [3]. The algorithm discovers the intervals in the graph one at a time. The first interval has the graph header node as its header. The algorithm proceeds by repeatedly including into the current interval each node that already has *all* its predecessors in the interval – this is what ensures the interval is the *maximal* subgraph. When no more progress is made, each node that has not all, but at least *some* predecessor in the interval becomes the header node of a new interval.

```
function find_intervals (graph) {
    intervals = []
    unprocessed_header_nodes = [graph.header_node]
    remaining_nodes = graph.nodes - {graph.header_node}

    while not empty(unprocessed_header_nodes) {

        // compute the interval with h as a header node
        h = unprocessed_header_nodes.pop();
        interval = Interval(h);
        intervals.append(interval);

        // add nodes to the interval while possible
        do {
            for node in remaining_nodes {
                if interval.contains_all(node.predecessors) {
                    remaining_nodes.remove(node);
                    interval.add_node(node);
                }
            }
        } while (a node was added to the interval)

        // compute more header nodes to process
        for node in remaining_nodes {
            if interval.contains_some(node.predecessors) {
                remaining_nodes.remove(node);
                unprocessed_header_nodes.append(node);
            }
        }
    }

    return intervals
}
```

Figure 9.2: The algorithm for interval partitioning

Hecht argues that the interval is a good representation of loops [38], because each interval corresponds to at most one loop. Additionally, intervals can be used to find a nesting order for loops. In contrast, using cycles is too fine a representation, since loops are not necessarily disjoint; and using strongly connected components is too coarse a representation, since they do not provide a nesting order [3].

Intervals provide a nesting order for loops through the so-called *derived sequence of graphs*. Let G_0 denote the original graph. Then it is possible to find the intervals of G_0 and collapse each interval $I(h)$ into a single node n. The predecessors of n are the predecessors of h that do not belong to the interval. The successors of n are the successors which are reachable from the interval but not part of the interval. Collapsing all the intervals gives a new graph G_1. Finding the intervals of G_1 and collapsing them gives a new graph G_2. This process can be iterated to a fixed point.[1] The sequence $G_0, G_1, ..., G_n$ is called the derived sequence of graphs.

This derived sequence is useful because the intervals of G_1 represent the loops of G_0 with the highest nesting level, the intervals of G_2 represent the loops with second-highest nesting level, and so on. It follows that if G_0 is the CFG of a program, the length of the derived sequence is proportional to the nesting level of loops in the program.

Once the derived sequence of graphs has been computed for some CFG, each interval contains at most one loop. Let $I(h)$ be an interval that may contain a loop. We define a *latching node* of an interval as any node which has h as a successor. It follows by the definitions we have given that an interval contains a loop if and only if it has at least one latching node. The loop structuring algorithm therefore computes the latching nodes; if there is none, the interval needs no further processing.

Determining the loop nodes

Given one or more latching nodes, the next task is to determine which nodes of the interval belong to the loop. This is necessary because while each interval may contain at most one loop, there is no guarantee that all the nodes in an interval are part of the loop. Figure 9.3 gives an example of a single interval where nodes 1 and 2 should be included in the loop, but nodes 3 and 4 should not.

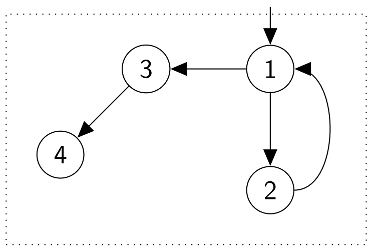


Figure 9.3: An interval containing a loop. Note that not all nodes in the interval are part of the loop.

[1] For some graphs, this iteration will reach a fixed point before the graph is collapsed into a single node; we then call the graph irreducible, and loop structuring fails. This occurs if the so-called canonical irreducible graph exists as a subgraph of the current graph [3]. We do not expect this to happen in programs compiled from Solidity, since the Solidity programming language does not contain a `goto` statement, and therefore cannot produce an irreducible graph.

To determine the loop nodes, Cifuentes describes an approach based on the reverse post-order numbering of nodes. In a reverse post-ordering scheme, nodes are visited in a depth-first search starting at the header node. Nodes are numbered on their *last* visit, and numbering starts with the highest number and counts down. In the example of Figure 9.3, the nodes are numbered in this way.

Cifuentes claims that in this numbering scheme, loop nodes are precisely the nodes with numbers between that of the header node and that of the maximal latching node. This certainly holds for the example of Figure 9.3.

Unfortunately when we implemented the algorithm, we found out that Cifuentes did not account for the fact that the reverse post-ordering of a graph is not unique; it depends on the order with which successors are visited. In Figure 9.3 we have preferred to visit the leftmost successor first during the depth-first search. If we had instead visited the rightmost successor first, we would have gotten the equally valid reverse post-ordering illustrated in Figure 9.4.

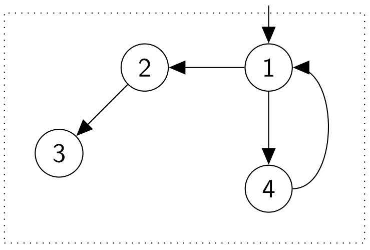


Figure 9.4: The graph of Figure 9.3 with an alternative (but equally valid) reverse post-ordering of the nodes.

Others have noticed this problem as well. To fix this issue, Yakdan et al. [25] propose adding an additional condition: for a node n to be included in the loop nodes, there must exist a path from n to the header h through a latching node.

In summary, the nodes of the loop are determined to be those with numbers between those of h and the maximal latching node, subject to the aforementioned constraint.

Determining the loop type

At this point it is possible to determine the type of the loop; a loop can be pre-tested, post-tested, or endless. Determining the type can be done by simple pattern-matching: a pre-tested loop has a conditional loop exit at the header node, a post-tested loop has a conditional loop exit at the maximal latching node, and an endless loop has neither, although it may still contain `break` statements.

However we found it simpler to delay this analysis until the back end. Therefore every loop is initially considered an endless loop. The readability improvements in the back end detect when an endless loop has a conditional loop exit as its header node. Such a loop is then transformed into a pre-tested loop.

Determining the loop follow node

The loop follow node is determined next. The follow node is the node to which `break` statements transfer control. If there is only one edge leading to some node *n* outside of the set of loop nodes, then *n* is simply made the follow node.

A more difficult case arises when there is more than one node reachable from inside the loop. In that case we call this a *multi-exit* loop. Such loops cannot be expressed in the Solidity language, since Solidity does not have a `goto` statement. Unfortunately, in practice the case arises anyway due to a compiler optimization.

A simple solution to this problem is to choose one node as the follow node, and to generate a `goto` for any other loop exits. The node with the smallest reverse post-ordering number should be chosen as the follow node, since the other nodes are reachable from it. This is the approach used by Cifuentes [3].

We have devised an alternative solution to this problem that results in fewer `goto` statements being produced. The general idea is that if more than one node can be reached from within the loop, then it may be possible to add one of these nodes to the set of loop nodes. By repeating this process until there is only one loop exit node left, the loop can be successfully structured. If adding a follow node to the loop is not possible, our decompiler can fall back to Cifuentes' approach and generate a `goto` statement.

We describe this approach in slightly more detail. Assume that we are in the process of structuring a loop, and that the loop has more than one exit node. Let OutReachable be the set of successors of loop exit nodes within the current interval. As long as the OutReachable set contains more than one node, the loop cannot be structured without `goto` statements. We therefore consider each $n \in$ OutReachable. If all predecessors of *n* are in the set of loop nodes, then *n* can be safely added to the set of loop nodes without changing the semantics of the program. We repeat this process as long as it is possible to do so, and as long as the OutReachable set contains more than one node.

To summarize the loop structuring algorithm of our decompiler, it is based on intervals analysis. Our decompiler processes each interval in the derived sequence of graphs, starting with intervals representing the most-nested loops. If the current interval contains a latching node, it contains a loop. The nodes belonging to the loop are determined using a reverse post-ordering numbering scheme, along with the constraint we described. The loop type is set to endless, to be fixed in the back end. Finally, the loop follow node is computed; if it is not unique, the just-described technique is used to increase the chances of fully structuring the loop.

9.2 STRUCTURING CONDITIONALS

Structuring conditionals involves finding the so-called *follow* of each conditional, which is the earliest point where the two branches of the conditional meet.

Finding the follow node of a conditional is important because it results in more readable high-level code after code generation. If a conditional has a follow, but it is not structured, then a `goto` statement is generated in one of the branches.

As a concrete example, consider the basic blocks of Figure 9.5. If the conditional at address `0x5a` is successfully structured, the basic block at address `0x80` is chosen as follow. In that case, code generation results in the code of

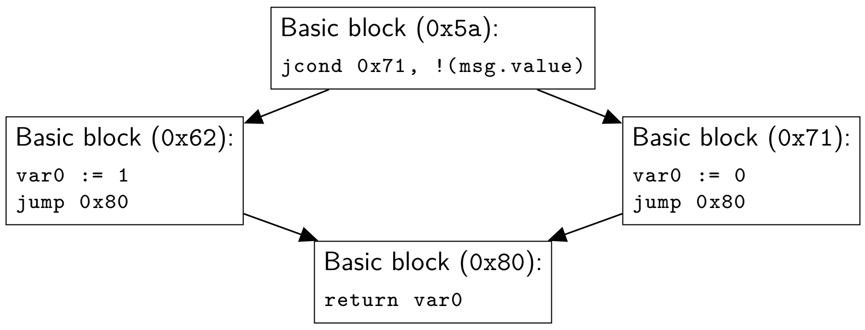


Figure 9.5: Basic blocks forming a conditional with a follow

Figure 9.6a. If structuring of conditionals is not performed, the code of Figure 9.6b is generated instead.

```
if (!msg.value) {
    var0 = 0;
}

else {
    var0 = 1;
}

return var0;
```

(a) Generated code with successful structuring of conditionals

```
if (!msg.value) {
    var0 = 0;
0x80:
    return var0;
}
else {
    var0 = 1;
    goto 0x80;
}
```

(b) Generated code without structuring of conditionals

Figure 9.6: The effects of structuring conditionals

To compute the follow of a conditional, the overall idea is to compute the set of basic blocks reachable from the true branch and from the false branch, take the intersection of these two sets, and pick the "earliest" node in this intersection as the follow.

Let c be a basic block that is a conditional, i.e., that ends in a `jcond` instruction. Let c_t and c_f be the first nodes along the true and false branches of c, respectively. We compute the follow of a conditional based on what we call the reach of a basic block. In the context of structuring conditionals, the reach of a basic block n is the set of basic blocks reachable from n, with some constraints, which we describe momentarily. Then let:

$$I = \text{reach}(c_t) \cap \text{reach}(c_f)$$

Then the follow of c is $\min(I)$, based on a partial ordering where $\text{BB}_1 \leq \text{BB}_2$ if there exists a path p from BB_1 to BB_2 where every basic block along p belongs to I.

When computing the reach, the outgoing edges of indirect jumps are not followed. This is because the set of possible successors of indirect jumps is an over-approximation of the actual successors, and thus an invalid follow may be found if such edges are followed. The consequence of not following such edges is that the computed reach is too small, but it is often still sufficient to find a follow node, thus improving readability. Additionally, edges back to the conditional c are not followed; such edges may occur as part of a loop. If such edges were followed, the loop would invalidate the partial ordering, as every basic block in I could reach every other basic block. If the conditional belongs to a loop, then edges leading out of the loop are also not followed, since loops and conditionals cannot overlap in this way in Solidity.

9.3 DUPLICATION OF TERMINATING BASIC BLOCKS

The Solidity compiler sometimes performs an optimization meant to reduce the size of the EVM bytecode. Unfortunately, this optimization sometimes makes the techniques for control-flow structuring that we have just described fail. Therefore undoing this optimization results in more readable code by producing fewer `goto` statements.

During compilation, the Solidity compiler may avoid producing duplicate code, instead inserting a jump to an existing address in the code. For example, if a Solidity function has multiple return statements, then the compiler may opt to output the function epilogue code only once, and insert a jump to this epilogue on all other occasions where the function returns. If our decompiler does not undo this optimization, it must output `goto` statements during code generation, since control-flow structuring fails.

The compiler optimization is undone in the following way. We look for any basic block that has no successors. This could be caused by a function return or by a terminating vmcall. If such a node is found, and it has multiple predecessors, then we duplicate the node until every predecessor has its own unique copy of the basic block as its successor.

This compiler optimization is not unique to the Solidity compiler. The same type of problem also arises, e.g., in the context of decompilation of x86 machine code; Yakdan et al. apply a similar technique in their decompiler to reduce the number of `goto`s in their output [25]. They note that it may be undesirable to duplicate large basic blocks, as it increases the output code size significantly.

10 BACK END

The back end generates high-level code in a Solidity-like language. It accomplishes this through a number of stages. First, the control-flow structuring information from the middle end is used to convert the intermediate representation into an abstract syntax tree (AST). The AST is then transformed and augmented to improve the final generated code. Finally, the AST is traversed to generate high-level code.

10.1 CONVERSION TO AN ABSTRACT SYNTAX TREE

It is be possible to generate high-level code directly from the intermediate representation of the middle end, without converting the program to an AST. However when we tried this, we found that keeping track of loops, conditionals, and follow nodes all at once, while attempting to generate code, was too difficult and error-prone.

Instead we convert the intermediate representation to an AST and generate code from that. Conversion to an AST can be done using a number of separate passes. This has the advantage that each separate pass is kept simple, facilitating the implementation and reasoning about each pass. Once the program is converted to an AST, code generation from the AST is straightforward.

The node types in our abstract syntax are given in Figure 10.1. The instructions and expressions are defined identically as in our intermediate representation.

```
Contract   ::= Function*
Function   ::= HeaderNode
HeaderNode ::= ASTNode

ASTNode    ::= Sequence(Instruction*)
             | IndirectJump(Loc)
             | IfElse(Condition, TrueNode, FalseNode, FollowNode)
             | Loop(HeaderNode, FollowNode)
             | Continue
             | Break
             | None

Loc        ::= Exp
Condition  ::= Exp
TrueNode   ::= ASTNode
FalseNode  ::= ASTNode
FollowNode ::= ASTNode
HeaderNode ::= ASTNode
```

Figure 10.1: A description of the nodes in our abstract syntax

Each function in the program is converted into an AST separately, through several distinct passes. First, every basic block is converted into a Sequence node. Then indirect jumps are removed from the Sequence nodes, being replaced by an edge to an IndirectJump node. Conditional jumps are also removed, being replaced by an edge to an IfElse node. The follow nodes of

IfElse nodes are set based on the structuring information from the middle end. After this, the structuring information is used to introduce Loop nodes. Once all the nodes are introduced, certain edges are removed from the graph according to the follow nodes of loops and conditionals. This ensures that the final result is a tree, i.e., that each node except the root has a unique parent.

10.2 READABILITY IMPROVEMENTS

After the program has been converted to an AST, the AST is modified and augmented in various ways to improve the readability of the final generated code.

One of the readability improvements is naming of variables. A pass over the AST assigns names to various variables, including function parameters, returned variables, and loop induction variables. Function parameters could for instance be named `param0`, `param1`, etc., while returned values can be named `result0`, `result1`, etc. Loop induction variables might be named `i`, `j`, etc. Storage variables can also be named.

Another pass recognizes accesses to variables of the `mapping` type so that more readable code can be generated for such mapping accesses. A Solidity mapping access is translated into EVM instructions as a storage access that is indexed by the SHA-3 hash of the mapping index concatenated with the mapping number. For example, the Solidity code `var0 = mapping2[var1]` could appear in our intermediate representation as:

```
mem[0x40:0x60] = var1;
mem[0x60:0x80] = 0x2;
tmp = sha3(mem[0x40:0x80]);
var0 = storage[tmp];
```

The pass for mapping recognition identifies this pattern and turns it into a statement involving a new type of expression called a `MappingAccess`. The above code then becomes:

```
var0 := mapping2[var1]
```

Separate passes recognize string- and array operations, which have similarly complex instruction patterns.

Some Solidity types are not 256 bits wide, which is the EVM word size. An example is the `address` type, which is 160 bits wide. All uses of such variables appear in the intermediate representation as a binary AND. For example, if `var0` is an address type, it appears in the IR as:

`var0 & 0xffffffffffffffffffffffffffffffffffffffff`.

Such large expressions are difficult to read, especially when multiple of them appear in a single line. A pass over the AST therefore discovers and removes such binary ANDs, replacing them with a cast. In our example, the output code thus becomes `address(var0)`.

Various other readability improvements are possible. For example, the code:

```
if (exp) {
}
else {
    statement;
}
```

is arguably more readable when converted to the following, equivalent code:

```
if (!exp) {
    statement;
}
```

Such minor transformations are applied in this phase of the decompiler.

10.3 CODE GENERATION

The code generation phase generates high-level code in a Solidity-like language. Code generation involves a pass over the AST. A symbol table is used to keep track of function names. A per-function symbol table keeps track of variable names.

Once the program has been converted to an AST, code generation is straightforward. For example, to generate code for a Contract node, the decompiler outputs "`contract DecompiledContract {`", generates the code for each function, then outputs "`}`". To generate code for a function, the decompiler first outputs its signature; this includes the function name, its parameter list, and its return values list, and could, e.g., be:

```
function func0 (uint param0) returns (uint, uint) {
```

The decompiler then generates code for the function header node, which recursively causes code generation to occur for the full function body. Finally it outputs a closing brace.

Code generation for a Sequence node simply involves generating code for each of its instructions, and then recursively generating code for its successor node. Code generation for a loop involves outputting "`while(`", generating code for the loop condition, and then outputting "`) {`", generating code for the body of the loop, and outputting the final "`}`". Code generation for the other node types, including IfElse, Continue and Break nodes, is similarly straightforward.

If a node is reached more than once, a `goto` statement is generated. This can happen if control-flow structuring fails. Reaching an IndirectJump node also produces a `goto` statement. All the possible successor nodes are then added to a set of pending nodes. Code is generated for these pending nodes by the end of code-generation for the current function.

It is necessary to generate labels for any nodes that may be the target of a `goto` statement. Such `goto` statements can occur if any IndirectJump nodes remain, or if either of the control-flow analyses fails. An initial pass computes which Sequence nodes could be the targets of `goto` statements based on node successor information. While generating code for such possible target nodes, a label, which includes the address of the node, is output before its instructions. For example, if the statement `goto var0;` has Sequence nodes with addresses `0x7f` and `0x131` as possible targets, then before generating code each of those nodes, the decompiler would output `0x7f:` and `0x131:`, respectively.

10.4 LOW-LEVEL CONCEPTS THAT REMAIN IN THE OUTPUT

The output from our decompiler would ideally be valid, compilable Solidity code. However in practice, producing compilable output is difficult, since some low-level details may remain in the code because of imprecision in the data-flow and control-flow analysis phases. Since the decompiler output is not always valid Solidity code, we refer to it as belonging to a Solidity-like language (SLL). In this section we describe the differences between Solidity and SLL, and argue about how remaining low-level details can be removed.

We have left the design of a type recovery system as future work due to time constraints. SLL therefore does not feature type annotations. Types have to be added to the output manually before SLL can be compiled with a Solidity compiler. Alternatively, the default type `uint256` could be used for each variable, since the generated code contains casts whenever a variable is used that is not of this type.

The notion of stack variables and the stack pointer do not exist in Solidity, but they sometimes remain in SLL. The decompiler uses stack flattening to remove stack variables whenever possible, but if indirect jumps remain, stack flattening may be impossible. Indirect jumps may remain due to expression propagation being too weak, or due to function identification failing, leaving the returns inside the function as indirect jumps. It is not always possible to resolve all imprecision, so all we can hope for is to reduce the occurrence of stack variables as much as possible.

A Solidity smart contract does not explicitly feature any loader code; rather, the loader code is inserted during the compilation process. Our decompiler does not currently attempt to remove the loader code. In fact, having the loader code explicitly available is desirable, since it shows which input values lead to which functions getting called.

11 IMPLEMENTATION

To investigate the practicality of our design, we have implemented a decompiler called DSol. This chapter gives an overview and some details of our implementation.

11.1 LANGUAGE OF IMPLEMENTATION

We chose Python 2.7 as our language of implementation. We found using a high-level language productive, although it did come with performance costs compared to a lower-level language such as C. We also found the multi-paradigmatic nature of Python useful, as it let us write in the style most convenient for each task, whether that was an imperative, an object-oriented or a functional programming style – we used all three during our development. We also found it a major drawback that Python is dynamically typed; static typing would have been very helpful in catching subtle bugs early.

In total, DSol comprises 7118 lines of Python code. The distribution of line counts across various modules are given in Figure 11.1. The figure shows that the majority of the programming effort went into the analyses in the middle end.

Module	**Line count**
Front end	699
EVM instruction representation	224
Parser	57
Conversion	418
Middle end	3556
Expressions	537
Intermediate Representation	210
Data-flow analysis tools	471
Expression propagation	251
Dead-code elimination	147
Function identification	474
Other analyses	652
Control-flow analyses	634
Fixed-point iteration	180
Back end	1010
AST representation	264
AST conversion	326
Readability	152
Code generation	268

Figure 11.1: A non-comprehensive overview of line counts for modules in DSol

We wrote each component from scratch; the only library we used that is not part of the Python standard library is the `pysha3` module for cryptographic hash computations.

11.2 DATA STRUCTURES

A number of data structures need to be represented during decompilation, including:

- EVM instructions
- Expressions
  - Literals
  - Unary operators
  - Binary operators
  - Variables
- IR statements
- Basic blocks
- AST nodes

We chose to represent all of these as objects. While this is not as efficient as using, say, tuples, or `structs` in a C-like language, we found using objects for our representation extremely convenient due to polymorphism and inheritance. For example, each type of expression has its own implementation of the `gen_code` method, so that the code generation module can work on a uniform interface, not having to worry about the type of expression it is dealing with.

11.3 SUCCESSORS AND PREDECESSORS

In the middle end of the decompiler, each function is represented as a CFG with basic blocks and edges between them. To represent these edges, we store successor information for a given basic block in the basic block itself. We also found it beneficial for performance to maintain predecessor information, rather the recomputing it each time it is needed. The successors and predecessors are represented as object references stored in sets, allowing $O(1)$-time membership tests.

11.4 ENUMERATING THE NODES OF A FUNCTION

Several of our analyses need to know which nodes belong to a function. Since successor information is updated by some analyses, the nodes of a function can change at any time. To compute the nodes belonging to a function, we perform a depth-first search starting at the function header node. As this is a major performance bottleneck, we found it helpful to cache the result, invalidating the cache entry whenever successor information for a function node is modified. We also found it important for performance to implement the depth-first search iteratively rather than recursively.

11.5 ALIAS ANALYSIS

Determining whether two variables may or must refer to the same location is called alias analysis [29]. This is required for many of the analyses in the middle end. For example, expression propagation involves discovering the possible definitions of a used variable v. This involves checking whether several variables may be aliases of v.

We have implemented a simplistic alias analysis. To answer whether two variables *may* be the same, the sound answer is "yes." To answer whether two variables *must* be the same, the sound answer is "no." Our analysis gives these answers by default.

If the compared variables have different types, a precise answer of "no" can be given. If the variables have literal offsets, such as when comparing `mem[0x40:0x20]` and `mem[0x60:0x20]`, then the analysis can also give a precise answer. However if the variables are indirect, such as `mem[var0:0x20]`, then our implementation gives up and falls back to the sound answer.

The only case where our implementation performs a more complex comparison is during the all-paths analyses, which keep track of changes to the stack pointer so that precise comparisons between stack variables is possible.

11.6 ALL-PATHS ANALYSES

Several of the analyses in the middle end require an all-paths analysis on the CFG. For expression propagation, all forward paths from the definition to the use of a variable must be checked for redefinitions. For dead-code elimination, all forward paths from the definition of a variable must be checked for uses of the variable. For internal function identification, all backwards paths the function header must be followed to find the possible values of the return address. We first implemented these analyses using techniques from optimizing compilers [29, 32].

For dead-code elimination, DSol needs to perform a *liveness analysis*. A variable is live at a program point if it may be used in the future. If the definition of a variable is not live, it can be eliminated. A technique for computing this liveness information based on Aho et al. [29] is as follows. The variables which are live going in and coming out of each basic block are maintained as sets. Data-flow equations are then defined, describing restrictions on the live-out set of one basic block in terms of the live-in sets of its successors. The analysis starts with a sound approximation of these sets. The idea is then to repeatedly improve this approximation until a fixed point is reached.

For expression propagation, DSol needs to perform a *reaching definitions* analysis, i.e., compute at which points a used variable may have been defined. Propagation is only possible if the definition is unique. This analysis can also be performed in terms of a fixed-point computation based on data-flow equations [29].

After implementing these techniques, we realized that this approach does not generalize well to the case when most data accesses are indirect, since we do not have access to a powerful alias analysis. As an example, our analysis determined that almost every stack variable is live, because *some* stack variable is used in the future, and the two may be equal.

We decided to abandon this approach. Instead DSol uses a less efficient approach that involves actually exploring every path. This has the advantage of being simple to implement. Additionally, the stack pointer can be kept track of along each path, which enables precise alias analysis of stack variables.

After writing three all-paths analyses, each with their own bugs, we decided to instead write a single module that is sufficiently general that it can be used for each of expression propagation, dead-code elimination and function identification. We found having a single, general module very useful, since it gave us fewer lines of code to maintain and debug.

We therefore wrote a single class, the `DefUseExplorer`, so called because it explores paths along the CFG while noting any definitions and uses of variables. The `DefUseExplorer` makes it possible to subscribe to definitions and uses of specific variables. The `DefUseExplorer` then explores all paths, following either successor or predecessor edges depending on the chosen direction. When a use or redefinition of a subscribed-to variable occurs, a callback is invoked.

The `DefUseExplorer` keeps track of changes to the stack pointer along its paths, which makes it possible to perform precise alias analysis of stack variables in different basic blocks. Without this, we could not determine if a variable such as `stack[sp-3]` is redefined in another basic block.

When the CFG is sufficiently imprecise, the number of paths to explore becomes too large to be practical. Rather than letting DSol run for an excessively long time, we have added a maximum number of basic blocks that the `DefUseExplorer` is allowed to explore. If the limit is reached, the `DefUseExplorer` gives up, and the safe option is taken for each analysis, so that propagation, elimination and function identification do not occur in this case.

11.7 ANALYSIS ORDER AND REDUNDANCY

In the middle end, we run a number of different analyses until reaching a fixed point. We have found that the order in which we run these analyses is important for the performance of the decompiler. This is because many of the expensive all-paths analyses perform far better once the precision of the CFG improves. Therefore, any analysis that is both cheap and can improve the precision of the CFG should be run to a fixed point first.

We therefore found it useful to group the analyses into two classes: cheap and expensive analyses. The cheap analyses typically run in less than a second even on large contracts, while the expensive analyses sometimes take tens of seconds to run. At first, only the cheap analyses are run repeatedly. Once a fixed point is reached, the expensive analyses are included in the fixed-point computation.

We created a number of redundant but very fast analyses to add to the category of cheap analyses. For example, dead-code elimination normally involves an expensive all-paths analysis to determine if a variable is used at a future program point. Sometimes, however, the variable is redefined within the same basic block. We can therefore create a very cheap, redundant intra-basic block dead-code elimination by simply taking the inter-basic block analysis and limiting it to a single block. The same is true for expression propagation.

The cheap analyses thus include intra-basic block propagation and elimination, constant folding, and assertion reconstruction. These analyses are all run until a fixed point is reached, and then the expensive analyses run; these include inter-basic block propagation and elimination, function identification, and stack flattening.

11.8 HANDLING OF DEPLOYMENT CONTRACTS

A contract is deployed to the network using a deployment contract. The contract-creating transaction results in the execution of the deployment contract. When the deployment contract is executed, it performs the work of the constructor of the contract. It then halts, returning the bytecode of the deployed contract that runs henceforth.

DSol recognizes deployment contracts by looking for a pattern that consists of a `haltreturn` vmcall that returns an area of memory loaded by the `coderead` vmcall. When a deployment contract is detected, DSol extracts the bytecode of the deployed contract and recursively decompiles it. During the decompilation of this second deployed contract, DSol retains the functions from the decompilation of the first deployment contract and adds these to the output. This is how the contract constructor is restored.

11.9 KNOWN ISSUES AND LIMITATIONS

Due to time constraints we have left certain problems unresolved in our implementation of DSol. The following list describes these limitations:

- The Solidity fallback function, which is the default function invoked if no external function is chosen, is not separated out from the loader code.
- Post-tested loops are not recognized and improved in the readability module.
- Correct but superfluous `continue` statements are generated at the end of some loops.
- Variable declarations are never output.
- String recognition is not implemented.
- Not every vmcall is represented in the output using syntactically valid Solidity code; for example, code generation for the `haltreturn` vmcall produces a `haltreturn` statement in the output, which does not exist in Solidity, rather than emitting inline assembly containing a `RETURN` instruction.
- Functions are not marked as external or internal in the output.
- Some of the readability transformations are currently implemented in the middle end rather than the back end.
- Storage variables are not named in the readability module.
- Casts are currently detected and accounted for during code generation rather than in the readability improval where this analysis conceptually belongs.

Note that all of these issues are not conceptually difficult to resolve; they simply require further engineering effort.

Additionally, during our evaluation we found that DSol fails to decompile a small percentage of contracts for a variety of reasons. We also found that DSol fails to structure certain conditionals that should be structured. We describe these issues and how to resolve them during the discussion of our results in Section 13.1 and 13.3.

12 EVALUATION

To evaluate our design and implementation, we have devised a number of experiments to investigate the robustness, correctness, and output readability of DSol. This chapter describes these experiments and presents our results. Interpreting and discussing the results is deferred until the next chapter.

12.1 EVALUATED METRICS

Existing research on decompilation uses a variety of metrics as the basis of evaluation. Schwartz [18] and Yakdan et al. [25] evaluate correctness by running the decompiled programs through a test suite with an expected output for each input. They also evaluate structuredness, a measure of how many `goto` statements remain in the output, since this decreases readability. Cifuentes [3] and Yakdan et al. [25] measure output compactness, which presumably reflects readability.

We have chosen to evaluate our decompiler based on all these metrics. Structuredness and compactness are useful metrics because they provide quantitative ways to evaluate the readability of the output of our decompiler. Correctness is crucial; without it, the decompiler output hurts rather than helps in understanding the semantics of the decompiled contract. Beside these three metrics, we additionally measure robustness, which is the ability of our decompiler to successfully decompile real smart contracts.

Several past works on machine-code decompilation [13, 25, 17] have performed a comparative evaluation with Hex-Rays, the "industry standard" of machine-code decompilation [13]. In our case there is no sufficiently mature decompiler for EVM bytecode that we can compare against.[1] Instead, we compare our results with those of existing machine-code decompilers when doing so is reasonable.

12.2 DATA SET

To evaluate machine-code decompilers, authors have either written their own small sample programs, or they have manually picked and downloaded open-source projects or malware samples. In contrast, the bytecode of every single smart contract on the Ethereum network is publicly available on the blockchain. We can use the bytecode of these contracts to evaluate the robustness, structuredness and compactness of our decompiler.

We are thus in the unique situation of being able to quickly download bytecode for thousands of real programs and evaluate our decompiler on these. As of December 2017, there were nearly a million smart contracts on the Ethereum blockchain [16]. For time reasons it is therefore not realistic to evaluate our decompiler on every existing smart contract; a subset of contracts must be chosen.

The online blockchain explorer Etherscan features a large collection of smart contract addresses with accompanying bytecode and Solidity source

[1] An early attempt at a decompiler for smart contracts, called Porosity, exists; however we were unable to successfully run the software, and the author does not provide an evaluation. Porosity is described in more detail in Section 15.6.

code [8]. We downloaded all 27593 smart contracts in this collection as of May 27, 2018. After removing contracts with duplicate bytecode, 27165 contracts remained, forming our input data set. We believe that this subset of smart contracts is representative of Ethereum smart contracts in general.

To better understand our data set, we have plotted the complexities of the contracts in Figure 12.1. The complexity is measured by the number of EVM instructions in the deployed contract. The figure shows that while many of the contracts have around 1500 to 3000 instructions, the data set includes some very small and some very large contracts as well.

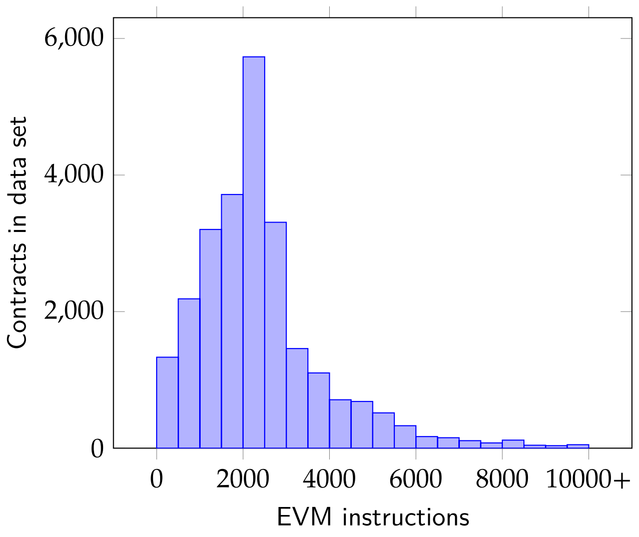


Figure 12.1: The number of EVM instructions per contract in our input data set

For evaluating correctness we have written a collection of sample programs with a series of inputs and corresponding expected outputs. These are described in more detail in Section 12.5

12.3 EXPERIMENTAL SETUP

To investigate the robustness, structuredness and compactness of the output of our decompiler, we decompiled every smart contract in our data set of 27165 contracts. We logged the pertinent details of each decompilation, including running time, the produced output or error message, the number of `goto` statements in the produced contract, and so on. The details were written to files on disk. We then analyzed this data. The results are presented in the following sections.

We decompiled the contracts on the machine whose details are given in Figure 12.2. The decompilation process is CPU bound, not memory bound. We therefore decompiled 8 contracts in parallel, since our machine has 8 threads of execution.[2] As some decompilations did not terminate, we set a maximum running time of 180 seconds; if decompiling surpassed this timeout, the process was terminated.

Decompiling the contracts that did not time out took a total of 187.91 CPU hours, spending an average of 25.63 seconds per contract. The distribution of the running times is illustrated in Figure 12.3. The figure shows that the ma-

[2] The machine only has 4 cores, so this may increase the average running time slightly.

Hardware	Description
Machine	Lenovo Y50-70
CPU	Intel Core i7-4700HQ, 2.40GHz
Memory	16 GB DDR3
Disk	256 GB SanDisk Ultra II SSD
OS	Arch GNU/Linux, kernel release 4.16.3-1-ARCH
Architecture	x86_64

Figure 12.2: Details of the machine used for our experiments

jority of decompilations finish within 30 seconds, although a non-negligible number of decompilations never finish, instead reaching the 180 second timeout.

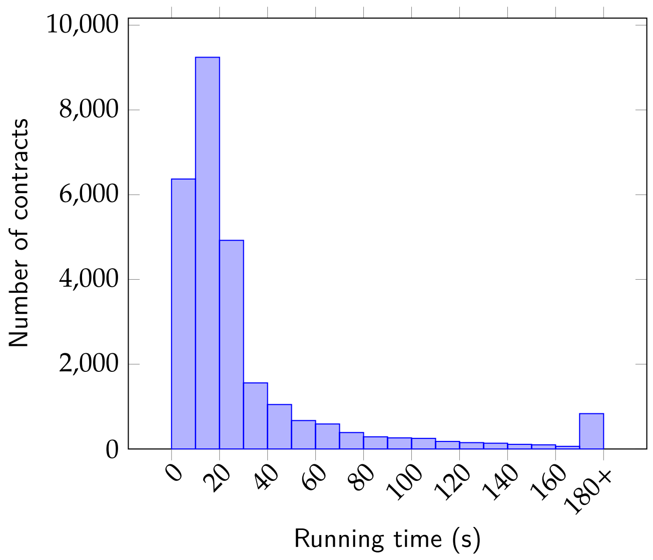


Figure 12.3: Running times of our decompiler

12.4 ROBUSTNESS

To investigate the robustness of our decompiler, we ran it on each contract in our input data set, noting whether the decompiler successfully produced an output, whether the decompilation timed out, or whether the decompiler failed to produce an output.

Our results are summarized in Figure 12.4. The figure indicates that our decompiler successfully produced an output for 92.18% of contracts in our data set.

It is natural to question whether our decompiler is only able to decompile less complex contracts, and fails or times out on larger ones. We therefore characterized the complexity of contracts in each category. We measure the complexity in terms of the length of the input bytecode, since this metric can be obtained even for contracts which could not be parsed. This characterization of complexities is illustrated in Figure 12.5.

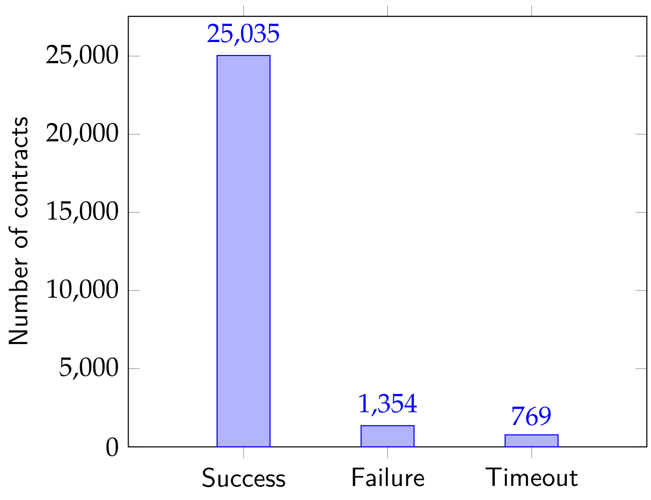


Figure 12.4: The outcomes of decompilation

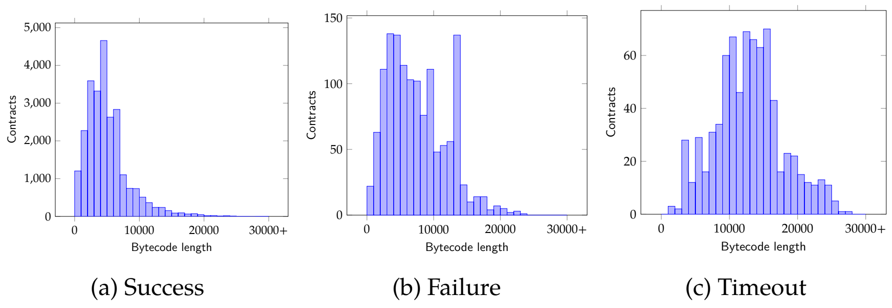


Figure 12.5: Distributions of bytecode lengths for decompilations which succeeded, failed, and timed out

The average bytecode lengths for decompilations which succeeded, failed, and timed out, are 5043, 7614, and 12860, respectively.

12.5 Correctness

When decompiling a program, it is crucial that the semantics of the program be preserved. Thus we need a way to check that the decompiler output is semantically equivalent to the input bytecode.

To investigate the correctness of the output of our decompiler, we wrote a number of smart contracts that exhibit common Solidity language features. We then wrote a number of tests for these contracts by manually reasoning about the behavior of each contract by examining the Solidity source code. That is, we determined a number of inputs and corresponding expected outputs for each contract. Thus if the Solidity compiler and our decompiler produce correct output, then the decompiled contract should return the expected output on each input.

Unfortunately our decompiler does not produce output that can be compiled back to executable bytecode; the biggest obstacle is that the decompiler does not perform type recovery at present. We therefore wrote an interpreter for the abstract syntax, which is the final representation of the decompiled program before high-level code is generated.

Note that because we interpret the AST, any errors in the final code generation phase are not caught by our correctness tests. However once the abstract syntax trees are produced, code generation is a straightforward traversal of the trees, with few opportunities to make mistakes. We have manually read the generated code of each test contract without spotting any mistakes in code generation.

In short, to evaluate correctness we decompiled each of our sample contracts, then interpreted the produced AST, providing the appropriate call data corresponding to an external function call. We then checked that the output returned upon halting was as expected, or alternatively, that the transaction was reverted when this was the expected behavior.

We wrote a total of 24 test contracts with a total of 120 tests. Our test contracts are described in Figure 12.6. We have run each test multiple times while compiling the test contracts both with and without compiler optimizations.

What is tested	**Contract name**	**Tests**
Simplest decompilation	Minimal	2
Function argument order	ArgOrder	1
Repeated call to internal function	Multicall	8
Call with many arguments	Multiargs	1
Return with many return values	Multiret	3
Four simple arithmetic functions	FourSimple	3
Mappings	Mapping	8
Arrays and constructors	Array	9
String handling	String	5
Endless loop	Endless	5
Simple loop	Loop	1
Nested loops	NestedLoops	1
A do-while loop	PostTestedLoop	4
Types smaller than 256 bits	SmallTypes	4
More types smaller than 256 bits	SmallTypes2	2
The greater-than operator	GT	2
Event logging	Log	1
Bitwise negation	Neg	3
Accessing contract storage	Storage	5
Non-commutative arithmetic	NonCom	17
Tail call optimization	TailCall	5
Nested conditionals	NestedIfElse	4
Conditionals with the same true- and false-node	IfElseSame	3
Language features used in esoteric ways	TryToBreak	23
Total		120

Figure 12.6: Contracts used to investigate correctness

Initially, not all of these tests passed. However we have made output correctness a high priority in the development of DSol; thus whenever a test failed, we made an effort to understand the source of the failure and fix it. Therefore all the tests pass.

Because we have fixed each correctness failure, and because our test contracts are relatively simple, we do not expect our result to generalize to practical contracts from the blockchain. We have planned an evaluation of correctness on more complex contracts, which we describe as future work in Section 14.9.

12.6 STRUCTUREDNESS

When our decompiler produces `goto` statements, readability is decreased [35]. It is therefore relevant to measure the extent to which our decompiler recovers control-flow structure, as a failure to do so results in `goto` statements being produced. We refer to functions with at least one `goto` statement as *unstructured functions*.

Structuredness in the context of decompilation has been measured by Schwartz [18] as the total number of `goto` statements in a collection of programs, and by Yakdan et al. [25] as the percentage of functions for which no `goto` statements appear.

The 25035 successful decompilations resulted in a total of 30902 `goto` statements being produced across 525945 functions.

We measured the number of contracts with and without at least one `goto` statement. This result is illustrated in Figure 12.7. The figure indicates that 12.47% of contracts contain a `goto` statement. In contrast, only 1.38% of func-

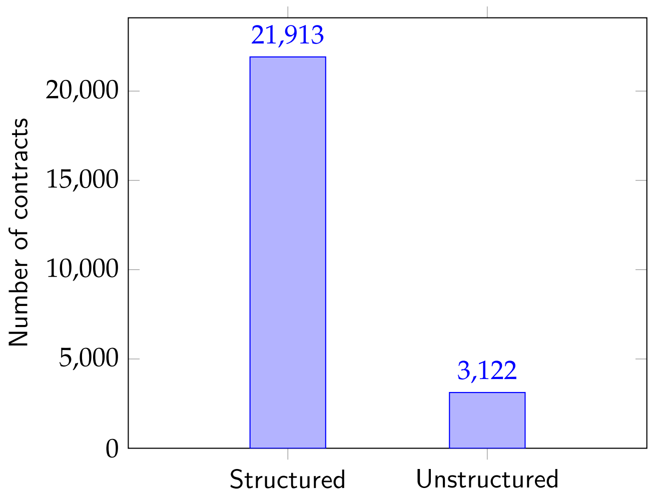
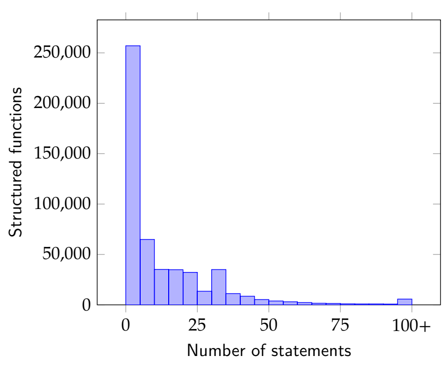


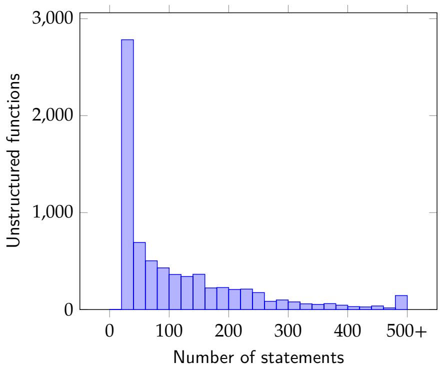
Figure 12.7: The number of contracts with and without `goto` statements

tions are unstructured. The average number of `goto` statements in these functions is 4.26.

To understand the context in which `goto` statements appear, we investigated the complexity of structured and unstructured functions, measured by the number of statements in the decompiled output. These are shown in Figure 12.8. The figure shows that the complexities of structured and unstructured functions are distributed differently. (Note that the axes in the figures have different scales.) Most structured functions are very small, having no more than 10 statements; in contrast, nearly all unstructured functions have 20 statements or more, and a sizeable amount have more than 200 statements.



(a) Structured functions



(b) Unstructured functions

Figure 12.8: A comparison of the number of statements in structured and unstructured functions.

On average, structured functions have 13.77 statements, while unstructured functions have 114.90 statements.

12.7 COMPACTNESS

Cifuentes [3] and Yakdan et al. [25] measure output compactness as part of their decompiler evaluation, with the idea that compactness is correlated with a higher abstraction level and therefore more readable code. They measure compactness in terms of a *reduction ratio*, which is the ratio between the number of instructions in the low-level input and the number of statements in the high-level output. Yakdan et al. [25] define:

$$\text{Reduction ratio} = \left(1 - \frac{\text{size of output}}{\text{size of input}}\right) \cdot 100$$

We therefore decided to measure compactness for our decompiler as the ratio between the number of EVM instructions in the input and the number of statements in the output.

Figure 12.9 shows how the reduction ratios are distributed across contracts. The figure shows that very few decompiled contracts have a reduction ratio lower than 70%.

Our compactness results for the average contract are summarized in Figure 12.10.

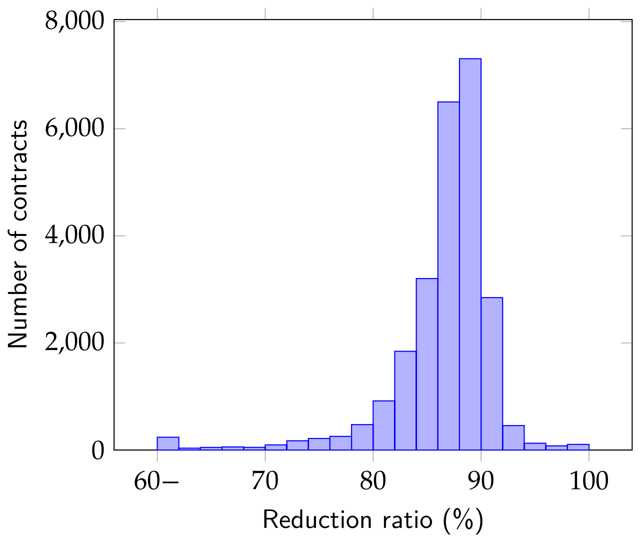


Figure 12.9: The distribution of reduction ratios across contracts

Measurement	**Result**
Average EVM instructions per contract	2379.38
Average statements per contract	318.67
Average reduction ratio	86.61%

Figure 12.10: Average results for compactness

13 DISCUSSION

In this section we interpret our results, compare them with those of related works, and discuss how to improve our decompiler.

13.1 ROBUSTNESS RESULTS

Our robustness results show that our decompiler produces an output for 92.18% of contracts in our data set. In contrast, Schwartz et al. [19] report their success rate on a per-function basis, producing a *recompilable* output for 82.2% of functions with their decompiler Phoenix, and for 77.5% of functions with the decompiler Hex-Rays. Note that this comparison is made with reservations, since there are differences between decompiling EVM bytecode and x86 machine code, and since Schwartz et al. only count recompilable output, which our decompiler never produces. Still, the success rates of Schwartz et al. indicate that a near-100% success rate cannot be expected.

We characterized the bytecode lengths of contracts for which decompilation succeeded, failed, or timed out. This is illustrated in Figure 12.5. Our characterization shows that contracts that cause decompilation failure are, on average, more complex than contracts that can successfully be decompiled. However there are also simple contracts which cause failures. The contracts which cause timeouts were, on average, more complex than contracts from the other two categories. This is expected, as the running time increases with contract complexity.

It is also worth understanding the cause of decompilation failures. During our experiments, our decompiler failed to decompile 1354 out of 27158 contracts. Of these failures, 1246 were assertion failures, while 108 were crashes. The assertion failures happen because we have inserted a number of sanity checks throughout the code: if some invariant no longer holds, we prefer that the decompiler halts without an output, rather than giving a wrong result. Producing a wrong output is still possible, but at least some of these cases can be immediately noticed and fixed. To give some examples, we check that no nodes become unreachable during AST conversion, that the successor and predecessors sets of nodes correspond with one another, and that `jump` and `jcond` statements only appear as the last instruction in a basic block.

This means that there is a trade-off between robustness and correctness; we could achieve a higher robustness result by disabling these assertions. We prefer that the decompiler produces no output to producing an incorrect one, however.

We have analyzed the error messages to understand why decompilation fails. The crashes are due to programming errors; this could, e.g., be from calling a method on an object that does not implement it, or using an object that is `None`. The assertion errors are the result of bugs. For example, the module that splits EVM instructions into basic blocks sometimes puts two terminating instructions (`jump`, terminating vmcalls, etc.) in one basic block. This is caught by an assert statement.

Other causes of assertion failures are a failure to find any valid successors for an indirect jump; encountering an unhandled EVM instruction; and the disappearance of nodes during AST conversion.

These robustness failures require engineering effort to fix; the cause must be understood and the decompiler must be modified accordingly. We do not expect fixing any of these issues to result in any major design changes. We believe so because we have already fixed a large number of these robustness issues without having to change the latest iteration of our design. We have not addressed the described issues due to time constraints.

We may also consider the decompilations that timed out as a type of failures. The distribution of Figure 12.3 indicates that few contracts take between 120 and 180 seconds to decompile, and then there is a larger amount that take 180 seconds or more, i.e., which timed out. This spike in contracts at 180+ seconds could indicate that some of the contracts in this category result in non-termination during decompilation. Indeed, after investigating and fixing a few such cases, we found this to be the case. In one such example, expression propagation had a bug that caused it to continue forever; the analysis used the same statement as a definition and a use, such that

```
var0 := (1 + var0)
```

was propagated forever, becoming

```
var0 := (1 + (1 + (1 + (1 + var0))))
```

and so on. We fixed this case, but addressing the remaining non-terminating cases is a matter of further engineering effort.

13.2 CORRECTNESS RESULTS

Our correctness results show that our decompiler successfully decompiles every one of the test programs we have written, and that the output is semantically equivalent to the original contract for our chosen inputs. While this result shows that our decompiler produces correct output for some small contracts, this result does not generalize to every contract on the blockchain, as our tests are not representative of large, complex contracts.

Section 14.9 describes our plans for a more thorough evaluation of the correctness of our decompiler, which we leave as future work due to time constraints.

13.3 STRUCTUREDNESS RESULTS

If we measure structuredness as the percentage of functions which are structured, we found in the previous section that 98.62% of decompiled functions do not have a `goto`. However our function identification analysis may duplicate some functions.[1] If the duplicated functions are of low complexity, and structuring low-complexity functions is easier, then this could bias our structuredness result if we measure it in terms of percentage of functions structured. It may therefore be more pertinent to consider the percentage of *contracts* structured.

Our structuredness results show that 87.53% of decompiled contracts did not have any `goto` statements. Thus for the vast majority of contracts, our control-flow analyses and function identification adequately recover high-level control-flow.

[1] The reason for this duplication is that our external function identification inlines any internal functions into every discovered external function. This means that some internal functions get duplicated during decompilation. Data-flow analysis may then modify these inlined functions, making each inlined function unique. This makes removal of duplicates impossible.

In comparison, Yakdan et al. [25] structured 91.2% of functions with their decompiler REcompile, while they structured 89.3% of functions with the decompiler Hex-Rays on the same data set. Schwartz [18] structured 99.83% of functions with his decompiler, Phoenix, while structuring 99.71% of functions with Hex-Rays on the same data set.[2]

Unlike Solidity source code, C source programs can contain `goto` statements. One might therefore assume that these machine-code decompilers cannot hope to structure 100% of functions. However in a recent work called "No More Gotos," Yakdan et al. [26] achieve a 100% structuring rate by using semantics-preserving transformations to structure CFGs that would otherwise be unstructurable.

That such a high structuring rate is possible even in the presence of `goto` statements indicates that we can realistically hope for a higher structuring rate in our own decompiler with further work.

It is worth considering the context in which our decompiler produces `goto` statements. Figure 12.8 shows that the distributions of complexities of structured and unstructured functions differ significantly. Structured functions are predominantly small, containing less than 20 statements. In contrast, unstructured functions almost never have less than 20 statements, and a very significant amount of functions have 200 statements or more. Together, these results imply that small functions are almost always structured, and large functions are rarely structured.

The causality behind this finding is unclear from the figure; it is plausible that complex functions are simply difficult to structure, but it is equally plausible that the failure to structure a function (e.g., due to a failure in function identification) is the cause of the complex output.

We also found that `goto` statements tend to cluster: on average, unstructured functions contain 4.26 `goto` statements. This, combined with the finding that many unstructured functions have 200 statements or more, implies that a significant fraction of these unstructured functions may be very difficult to read, which is a cause for concern.

Since the Solidity programming language, which the contracts of our data set were compiled from, does not have a `goto` statement, one might wonder why our decompiler outputs `goto` statements in the first place. We have manually investigated some of the contracts that could not be structured. Our investigation found two different causes for a failure to structure a function.

The first cause of `goto` statements is when structuring of conditionals fails. When a follow node could not be found, but the two branches of the conditional *do* meet somewhere, then one of the branches includes the code of the follow node, and the other branch contains a `goto` to the follow node, which is in the middle of the first branch. We do not expect such structuring failures to happen at all; the failures imply that our implementation contains a bug either in structuring of conditionals or in AST conversion. For time reasons, we leave investigating and fixing this issue as future work.

The second cause of `goto` statements is when function identification fails. If a function is not identified, and it is called more than once, then each function call becomes a `goto` to the function header, and the function `return` statements are represented as indirect jumps, becoming `goto` statements with some variable as the location.

[2] This discrepancy in the structuredness results of Hex-Rays between Yakdan et al. and Schwartz is surprising. The discrepancy is likely due to different input data sets: Yakdan et al. used malware samples and a web browser, while Schwartz used the GNU coreutils suite of programs, which is plausibly less difficult to structure.

Function identification may fail if the possible values of the return address cannot be determined. This can happen if the CFG has insufficient precision, or if our analysis is not powerful enough. This issue is more difficult to fix, as it does not have a single cause. More of these cases can be handled by careful investigation and improval of our analyses, letting them handle more special cases.

13.4 COMPACTNESS RESULTS

Our experiments show that our decompiler achieved a reduction ratio between 80% and 95% for the vast majority of contracts, with an average reduction ratio of 86.61%.

In comparison, Cifuentes [3] achieved an average reduction ratio of 76.25% across 10 small example programs. However, since her decompiler translates from 80286 to C, and her inputs are much smaller on average, the comparison may not be particularly apt.[3]

We have manually examined some of the contracts with low reduction ratios. This analysis showed that a major cause of low reduction ratios is when function identification fails; this creates complex functions with many statements, and complex functions are in turn harder for our data-flow analyses to analyze and simplify due to the large number of possible paths in the CFG. Improving our structuredness results as described in the previous section is therefore the first step we would take to improve compactness as well.

[3] We were unable to find reduction ratios from other authors, since they tend to not provide raw numbers, but instead provide a percentage of functions on which they have a better reduction ratio than, e.g., Hex-Rays.

14 FUTURE WORK

14.1 SYMBOLIC EXECUTION

Implementing symbolic execution would allow DSol to reason about the possible values of variables. This would allow further improvements in the precision of the CFG by reasoning about possible targets of indirect jumps. Improving the precision of the CFG has a dramatic effect on the decompiler output, since it makes most analyses in the middle end both more efficient and effective. Symbolic execution would therefore be a valuable addition to our decompiler.

14.2 INTERMEDIATE REPRESENTATION ON SSA FORM

In his PhD thesis, van Emmerik [22] argues for the advantages of using static single-assignment (SSA) form as the intermediate representation during decompilation. He argues that SSA form makes several of the analyses simpler to implement and more performant. It would be interesting to change our intermediate representation to be on SSA form throughout the middle end of the decompiler, and then investigate the effects on performance, simplicity of implementation, and analysis effectiveness.

14.3 ALIAS ANALYSIS

Currently, our measurements indicate that the all-paths analyses in the middle end are one of the main performance bottlenecks in DSol. Our current implementation explores every path in the CFG, which is excessively slow. As described in Section 11.6, expression propagation and dead-code elimination can instead be implemented using liveness- and reaching definitions analyses using a fixed-point computation based on data-flow equations. This performs better, but is too inaccurate without a powerful alias analysis.

We therefore would like to design and implement a more powerful alias analysis that can reason about aliasing of stack variables that reside in different basic blocks. Being able to more effectively reason about the aliasing of memory- and storage variables would additionally allow further propagation and elimination, thus producing more readable output.

14.4 FUNCTION SIGNATURE GENERATION AND RECOGNITION

It would be useful to have a way to match decompiled functions to known snippets of source code. If a decompiled function is matched to a snippet of known source code, that allows our decompiler to recover the original function name, descriptive variable names, types, and so on.

As an example, many smart contracts make use of a library for performing "safe math" that checks for overflows when performing arithmetic. If a decompiled function could be identified as the `add` function from this library, it could be named appropriately, easing the understanding of the decompiled contract.

This signature detection can easily be accomplished for external functions, because every external function has a corresponding hash that consists of four bytes. These bytes are used to select the function in the loader code. We envision a tool that processes a large database of Solidity source code, such as the collection of contracts we used for our evaluation. The tool then computes the hash of each function signature, creating a database mapping from a four-byte hash to known source code. Our decompiler could then use this database to recover function and variable names, and parameter and return value counts.

14.5 TYPE RECONSTRUCTION

Our decompiler currently uses ad-hoc pattern matching to recognize accesses to mappings and arrays. It also recognizes some binary AND operations as casts, e.g., to an address type. However structs are not recovered, and neither are string operations. As it is based on pattern matching, our approach is unprincipled, and frequently fails to recover information.

The ability to recover types in a principled manner would improve the readability of the decompiler output by increasing its compactness and clarity. Additionally, type reconstruction is required to produce output that is recompilable.

We describe type reconstruction techniques in more detail in Section 15.8. In short, current techniques are based on generating and solving constraints. We leave the design and implementation of a principled type recovery system as future work due to time constraints.

14.6 COMPILABLE OUTPUT

Currently our decompiler output is not recompilable, and we have not made any effort to make it so. Ideally, however, the output would be valid Solidity code for as many contracts as possible. Producing compilable output would allow us to evaluate the correctness of the final output of the decompiler, rather than of the AST as we currently do through interpretation.

The most notable obstacle to producing compilable output is the lack of type recovery. Additionally, all low-level details must be removed from the intermediate representation. This requires that our analyses be effective, which is principally a matter of engineering effort. In the worst case, the decompiler could produce inline EVM assembly for any low-level details that remain.

14.7 INTERACTIVITY

Our decompiler is currently a terminal-based application that produces a single, static output. When the decompiler is used as a tool for reverse-engineering a contract, modifying the output to, e.g., rename a variable, has to be done manually in a separate editor.

Our decompiler could offer an alternative graphical user interface which lets the user rename functions and variables after decompilation, refactor the output in various ways, explore graphs of the CFG and AST of the output, and so on. This would increase the utility of our decompiler in the reverse-engineering process, at the cost of further engineering effort.

14.8 FURTHER EVALUATION OF READABILITY

We have attempted to quantify certain aspects of readability in our evaluation. However readability is an inherently subjective matter. It would therefore be useful to carry out an evaluation of readability based on user feedback. Such an evaluation would involve a number of participants with varying degrees of experience reverse-engineering low-level code.

The participants could be given the decompiler output for a realistic smart contract, with the task of understanding the behavior of the smart contract. Participants' understanding could be quantified by having them answer a series of questions about the behavior of the contract. We could then compare the degree of understanding, as well as the time taken for analysis, with similar data for participants given the original source code rather than the decompiler output.

It would also be valuable to identify which aspects of decompilation most hinder and help the analyst's understanding, in order to understand the shortcomings of DSol and decide which areas to focus further research and development on. For example, does output structuredness have a high impact on readability, or should our efforts be concentrated elsewhere? What about function identification? Such questions could be answered through questionnaires and user interviews.

Relatedly, a user-driven evaluation of machine-code decompilers was carried out by Yakdan et al. [24]. However Yakdan et al. perform a comparative evaluation between three different decompilers. As we have no other decompiler to compare with, we would instead compare the readability with that of the original source code.

14.9 FURTHER EVALUATION OF CORRECTNESS

We evaluated the correctness of DSol on a set of small contracts exhibiting a variety of Solidity language features. This involved writing an interpreter for the abstract syntax produced in the back end of DSol. We manually picked inputs and deduced expected outputs for each contract and ensured that the abstract syntax produced by DSol behaved as expected. The main limitation of this evaluation is that the contracts we used for testing are not representative of real smart contracts, since they are not as complex.

We have therefore planned a more thorough evaluation of correctness. The blockchain contains every transaction ever made to every existing smart contract. This provides us with an enormous set of sample inputs to smart contracts. By running each transaction on the EVM, we can log the returned data and caused side effects. In other words, we can convert *every* transaction on the blockchain into a correctness test for DSol.

Carrying out such an evaluation involves a variety of further work. Transactions must be extracted from the blockchain. The return values and side effects of each transaction must be determined. Our AST interpreter must be modified to log side effects. The interpreter must also be modified to set up the environment such that it reflects the exact state of the blockchain at the time of each transaction. For example, if a contract retrieves the balance of an account, the corresponding vmcall must return the balance at the time when the transaction was originally made.

15 RELATED WORK

15.1 PIONEERS OF DECOMPILATION

Decompilation has a surprisingly long history. A decompiler from machine code to an Algol-type language was written as early as 1960 [3]. Early decompilers were written in an unprincipled manner, relying on pattern recognition. They also relied heavily on manual intervention to handle difficult instructions, partitioning the code into functions, determining arguments to a call, etc.

Housel's PhD thesis from 1973 describes a more principled approach to decompilation, borrowing techniques from compiler-, graph-, and program optimization theory [39]. Housel's decompiler has three stages. In the first stage, the input assembly is translated to a machine-independent intermediate representation, and a control-flow graph is built. In the second stage, redundant instructions are removed and loops are detected. In the third stage, high-level code is generated.

A decompiler was built according to Housel's approach. The main limitations of this work are that the input language, MIX assembler, provides more information than machine code, and 12% of instructions required manual intervention [3].

In the 80s and 90s more decompilers were designed and built. They were used for porting assembly programs from one generation of machine to another, to recover lost source code, and to modify existing binaries. The produced high-level code was also used to ease maintenance and function as documentation [3]. However decompilers of this time period required significant amounts of manual intervention.

15.2 REVERSE COMPILATION TECHNIQUES

Cristina Cifuentes' PhD thesis, "Reverse Compilation Techniques" [3], is the first work to systematically describe the known techniques and methodology for decompilation. Written in 1994, Cifuentes was first in tackling decompilation of realistic assembly code without simplifying assumptions or manual intervention. Her decompiler, called dcc, translates from Intel 80286 assembly to C.

Cifuentes' design can be considered a refinement of Housel's. The decompiler is split into three phases: a front end, a middle end, and a back end. This is illustrated in figure 15.1.

The front end parses the input assembly, generates an intermediate representation of it, and also creates a control-flow graph. The middle end is responsible for removing low-level details from the code. It first performs data-flow analyses to propagate expressions and eliminate dead code. The middle end then performs control-flow analyses to discover conditionals and loops. Finally, the back end generates high-level code.

Cifuentes provides a principled approach to decompilation, and she systematically describes the various techniques and algorithms needed in a modern decompiler. Moreover, almost all modern decompilers that we are aware of use a design reminiscent of Cifuentes'. We have therefore used her thesis

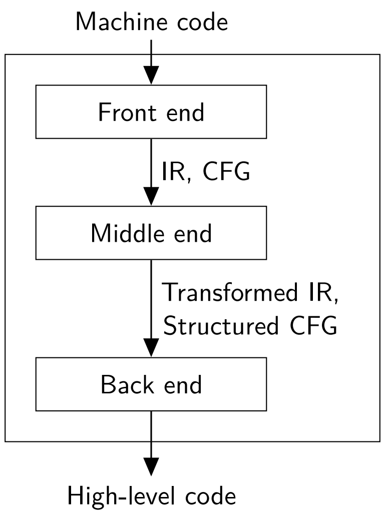


Figure 15.1: Overall design of Cifuentes' decompiler.

as our main source during the design and implementation of our own decompiler. Our own design uses Cifuentes' as its starting point.

15.3 STATIC SINGLE-ASSIGNMENT FORM

The PhD thesis of van Emmerik [22] investigates the use of single static assignment (SSA) form in decompilation. The techniques from the thesis were tested in an open-source decompiler, Boomerang, work on which began in 2002 [13], while the thesis itself is from 2007.

van Emmerik highlighted the deficiencies of machine-code decompilers of the time: they relied on calling conventions to recover parameters and return values, and they handled indirect jumps poorly. These problems become more tractable when the program is in SSA form. In particular, data-flow analyses such as expression propagation become far simpler; such analyses must reason about the possible definitions of a variable, which is unique under SSA form.

Additionally, earlier decompilers either performed no type analysis at all, or they used ad-hoc reasoning, propagating known types from library calls. van Emmerik investigated different approaches to type analysis, though he concluded that "much work remains before type analysis for machine code decompilers is mature."

15.4 HEX-RAYS

Hex-Rays is the *de facto* industry standard machine-code decompiler [18, 41], focusing primarily on decompilation from x86 to C, although many other architectures are supported. Hex-Rays is not free software; decompiler modules cost thousands of dollars. Accordingly, information about the internals of Hex-Rays is very sparse, although an old technical report from 2008 is publicly available [12] from which some details can be gleamed.

Hex-Rays appears to largely follow a design similar to that described by Cifuentes, with a data-flow analysis phase followed by a control-flow analysis phase. After these phases, Hex-Rays applies program transformations to improve the readability of the code. It then performs type analysis. Finally variables are named and the output code is generated.

15.5 PHOENIX

Schwartz described the design and implementation of a decompiler, called Phoenix, in his PhD thesis from 2014 [18] and a related paper [19]. Schwartz proposed the use of decompilers to increase the scalability of binary program analysis; he argues that some analyses can be sped up significantly by first recovering high-level abstractions, and that many existing program analysis techniques are designed to work on source code rather than low-level code.

The Phoenix decompiler translates from x86 to C, producing compilable output. It features a staged design similar to Cifuentes', but includes a type recovery phase based on a component called TIE (Type Inference on Executables) [41]. Phoenix also uses a new technique for control-flow analysis that can proceed with loop structuring even in the presence of gotos.

Schwartz argues that correctness should be an important metric in evaluation of a decompiler; previous machine-code decompilers did not evaluate correctness, except by manual inspection of the output for a few example programs. Schwartz devised an experiment to evaluate the correctness of his decompiler, which involved decompiling a set of programs with an accompanying test suite. We have used this as inspiration for evaluating correctness in our own decompiler.

15.6 POROSITY

Porosity [20] is the only existing public decompiler for Ethereum smart contracts. It is open source. Porosity represents an early attempt at smart contract decompilation, translating directly from EVM instructions to Solidity code. The control-flow graph of the program is recovered using symbolic execution. External functions are identified from the bytecode using heuristics.

Porosity has no intermediate representation, nor does the decompiler use data-flow analyses or loop recovery to improve the output. The document describing Porosity [20] does not provide an evaluation of the software.

We attempted to perform a comparative evaluation between Porosity and DSol. However we could not run Porosity on any smart contracts without it crashing. Fontein [11] likewise failed to evaluate Porosity due to a lack of robustness.

15.7 RETARGETABLE DECOMPILER

Previous decompilers have largely focused on decompilation from a specific architecture to a specific language. In contrast, the Retargetable Decompiler is a project that aims to support decompilation from machine-code of a large number of architectures into various high-level languages. The project is described by Křoustek in his thesis [13]. To decrease the amount of manual work per input architecture, an architecture description language is used to automatically convert between machine code and intermediate representation. It is also worth noting that the decompiler delegates various of its analyses to the LLVM compiler infrastructure.

15.8 TYPE RECONSTRUCTION

When low-level code is decompiled to a typed high-level language, type reconstruction is required to produce recompilable output.

Early machine-code decompilers such as Cifuentes' [3] typed every variable as `int` by default, inserting casts as needed. Early decompilers also used prototypes of known standard-library functions to infer more types.

Mycroft [46] describes an approach for type reconstruction during decompilation that is based on generating and solving type constraints. Mycroft also describes techniques for recovering recursive data types, and types of recursive functions. Lee et al. [41] describe a similar approach based on constraint generation and solving, which is made more principled using a formal type system based on lattice theory, ensuring that type reconstruction is conservative and sound.

15.9 EVOLVING EXACT DECOMPILATION

Schulte et al. [17] recently proposed a novel approach for machine-code decompilation. The general idea is to combine snippets of source code, compile them, and check to which extent the produced machine code matches the program that is being decompiled. Source code snippets are combined from a large database of code using an evolutionary algorithm. One advantage of this approach is that if such an evolutionary decompiler can evolve source code that compiles down to the exact bytecode of the decompiled program, then semantic equivalence is guaranteed.

15.10 LOOP STRUCTURING IN DECOMPILATION

The technique for loop structuring based on intervals analysis was originally used by Cifuentes [3]. We have described this approach extensively in Section 9.1, since we have used it in DSol. As we mentioned in that section, standard intervals analysis has the limitation that it is unable to structure irreducible graphs.

Rather than giving up on structuring such irreducible graphs, van Emmerik [22] proposed a technique called *iterative refinement*; the idea is to pick and remove an edge from the graph, deliberately choosing to emit a `goto` statement for that edge. The removal of the edge then allows a technique such as interval analysis to structure the rest of the graph after all. The effect is that the total number of `goto` statements is reduced, even if structuring the graph cannot fully succeed.

Yakdan et al. [26] propose a different technique based on program transformations that preserve the semantics of the program while eliminating `goto` statements. Unlike previous works, Yakdan et al. manage to produce decompiler output without a single `goto` statement.

15.11 FUNCTION IDENTIFICATION

Function identification involves separating parts of a binary program into functions. This involves discovering the beginning and end of a function. Function identification is important during decompilation, since it affects the readability of the produced high-level code [6]. The problem of function identification can alternatively be phrased as constructing a call graph for a binary program.[1] Function identification has applications in other areas of binary code analysis; for example, it is a necessary first step in protecting binary applications with control-flow integrity [28].

Durfina et al. [6] propose a technique for function identification in their decompiler. Their decompiler is "retargetable," which means that it is designed to support multiple architectures. It must therefore identify functions in a general manner. Durfina et al. describe a top-down approach, which splits the whole program into smaller chunks, each representing a function. They also describe a bottom-up approach, in which instructions are merged into blocks until each block represents a function. Both approaches assume that the decompiled program contains call instructions. It uses the targets of these call instructions to split or merge the blocks of the program.

Another common approach to function identification is based on signature recognition; machine-code functions often contain a function prologue that, e.g., saves registers to the stack, and this sequence of machine code can be recognized [31]. Bao et al. [2] extend this approach; they use machine-learning to identify bytecode sequences that commonly appear in function prologues, thus identifying where each function begins. This approach has the advantage that it can automatically be adapted for new compilers and architectures. However false positives are a concern if function identification leads to changing the semantics of the program.

15.12 ANALYSIS OF EVM BYTECODE

Various tools exist for analyzing EVM bytecode. Ethersplay [10] is an EVM disassembler which can perform some rudimentary analyses on the disassembly; it uses pattern matching to separate out external functions from the loader code. It uses a value set analysis of stack variables to resolve indirect jumps and build a CFG. Mythril [14] is another tool for analyzing Ethereum smart contracts. Mythril employs concolic execution, a combination of symbolic and concrete execution, to discover vulnerabilities.

Luu et al. [42] built a symbolic execution tool for Ethereum smart contracts called Oyente. Luu et al. use Oyente to automatically discover vulnerabilities in deployed smart contracts. Similarly, Nikolic et al. [16] utilize symbolic execution to automatically discover vulnerabilities that occur across multiple transactions.

[1] A call graph is a graph that represents program functions as nodes and possible flows between functions as edges

16 CONCLUSION

We have presented our design and implementation of DSol, a decompiler for Ethereum smart contracts.

We described our design, which divides decompilation into three phases: a front end, a middle end, and a back end. The front end translates the EVM bytecode into an intermediate representation. The middle end applies a number of analyses until reaching a fixed point in order to raise the abstraction level of the program. The middle end also performs control-flow analysis to recover high-level control-flow structures. The back end produces high-level code after converting each function to an abstract syntax tree.

The biggest challenge in designing our decompiler was the lack of information inherent in EVM bytecode. This gave rise to a number of problems that our design had to overcome. Our design solves the lack of access to a precise control-flow graph through iterating data-flow analyses to a fixed point. Our design solves the difficulty of identifying functions in the EVM bytecode by using a heuristic for easily-identifiable external functions, and a more elaborate analysis based on safety conditions for internal functions.

Finally, our design overcomes the difficulty of alias analysis when most data accesses are indirect by using a number of techniques: it keeps track of the stack pointer during all-paths analyses, it applies stack flattening, and it consolidates changes to the stack pointer within each basic block.

We implemented a decompiler, which we call DSol, according to this design. We proceeded to evaluate our design and implementation through experiments; our metrics of choice were robustness, correctness, structuredness, and compactness. We found that DSol successfully produced an output for 92.18% of contracts in our data set. We also found that DSol succeeded in structuring 87.53% of decompiled contracts and 98.62% of functions, producing an output without `goto` statements. Additionally, we found that DSol is capable of producing correct output for simple contracts exhibiting a variety of Solidity language features.

All in all, our experiments showed that DSol is capable of decompiling and structuring practical smart contracts from the blockchain.

REFERENCES

PRIMARY LITERATURE

[1] Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer.

[2] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. (2014). Byteweight: Learning to recognize functions in binary code. USENIX.

[3] Cifuentes, C. (1994). *Reverse compilation techniques*. Queensland University of Technology, Brisbane.

[4] CoinMarketCap. Cryptocurrency market capitalizations. `https://coinmarketcap.com/`. Accessed: 2018-06-02.

[5] Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. (2016). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer.

[6] Durfina, L., Kroustek, J., Zemek, P., and Kabele, B. (2012). Detection and recovery of functions and their arguments in a retargetable decompiler. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 51–60. IEEE.

[7] Ethereum Project. `https://www.ethereum.org/`. Accessed: 2018-06-02.

[8] Etherscan. Ethereum contracts with verified source codes. `https://etherscan.io/contractsVerified/`. Accessed: 2018-05-26.

[9] Etherscan. Ethereum market capitalization and supply statistics. `https://etherscan.io/stat/supply`. Accessed: 2018-06-02.

[10] Ethersplay. Evm disassembler. `https://github.com/trailofbits/ethersplay`. Accessed: 2018-06-14.

[11] Fontein, R. (2018). Comparison of static analysis tooling for smart contracts on the evm.

[12] Guilfanov, I. (2008). Decompilers and beyond. *Black Hat USA*.

[13] Křoustek, J. (2014). *Retargetable Analysis of Machine Code*. PhD thesis, PhD thesis, Brno, FIT BUT.

[14] Mythril. Security analysis tool for ethereum smart contracts. `https://github.com/ConsenSys/mythril`. Accessed: 2018-06-14.

[15] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[16] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*.

[17] Schulte, E., Ruchti, J., Noonan, M., Ciarletta, D., and Loginov, A. Evolving exact decompilation.

[18] Schwartz, E. J. (2014). *Abstraction Recovery for Scalable Static Binary Analysis*. PhD thesis, Carnegie Mellon University.

[19] Schwartz, E. J., Lee, J., Woo, M., and Brumley, D. (2013). Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, volume 16.

[20] Suiche, M. (2017). Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF CON*, 25.

[21] Underhanded Solidity Coding Contest. `http://u.solidity.cc/`. Accessed: 2018-06-03.

[22] Van Emmerik, M. J. (2007). *Static single assignment for decompilation*. University of Queensland.

[23] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32.

[24] Yakdan, K., Dechand, S., Gerhards-Padilla, E., and Smith, M. (2016). Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 158–177. IEEE.

[25] Yakdan, K., Eschweiler, S., and Gerhards-Padilla, E. (2013). Recompile: A decompilation framework for static analysis of binaries. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 95–102. IEEE.

[26] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. (2015). No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*.

[27] Etherscan. Ethereum accounts and contracts. `https://etherscan.io/accounts`. Accessed: 2018-06-03.

SECONDARY LITERATURE

[28] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4.

[29] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, Principles, Techniques*, volume 7.

[30] Allen, F. E. (1970). Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM.

[31] Andriesse, D., Slowinska, A., and Bos, H. (2017). Compiler-agnostic function detection in binaries. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 177–189. IEEE.

[32] Appel, A. W. (1997). *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press.

[33] Chow, F., Chan, S., Liu, S.-M., Lo, R., and Streich, M. (1996). Effective representation of aliases and indirect memory operations in ssa form. In *International Conference on Compiler Construction*, pages 253–267. Springer.

[34] Cocke, J. (1970). Global common subexpression elimination. In *ACM Sigplan Notices*, volume 5, pages 20–24. ACM.

[35] Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

[36] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. (2016). Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59.

[37] Gencer, A. E., Basu, S., Eyal, I., van Renesse, R., and Sirer, E. G. (2018). Decentralization in bitcoin and ethereum networks. *arXiv preprint arXiv:1801.03998*.

[38] Hecht, M. S. (1977). *Flow analysis of computer programs*. Elsevier Science Inc.

[39] Housel, B. C. (1973). A study of decompiling machine language into high-level machine independent languages.

[40] Kinder, J., Zuleger, F., and Veith, H. (2009). An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 214–228. Springer.

[41] Lee, J., Avgerinos, T., and Brumley, D. (2011). Tie: Principled reverse engineering of types in binary programs.

[42] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM.

[43] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[44] Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer.

[45] Miecznikowski, J. and Hendren, L. (2002). Decompiling java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*, pages 111–127. Springer.

[46] Mycroft, A. (1999). Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer.

[47] Solidity Documentation, release 0.4.25 (2018).

[48] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp.

APPENDIX: OBTAINING AND RUNNING THE SOFTWARE

The source code of DSol can be retrieved at:

`http://52.24.122.73/dsol.zip`

The archive has the password:

`decompilationofethereumsmartcontracts`

and the SHA-256 hash:

`32ece594b34ddf55e8a0053d8d165dfa6c91724529dc4564134d2afa58ac6e07`

To run DSol, navigate to the `source/` directory and run the file `main.py` using Python 2.7:

```
$ cd dsol/source
$ python2.7 main.py
Usage: main.py <filename>
```

Note that to run, DSol requires the `pysha3` module to be installed for Python 2.7. The first and only argument to the decompiler is the name of the file to decompile. This can be a `JSON` file created by the `solc` Solidity compiler, or a file containing hex-encoded bytecode. The bytecode can either be deployment bytecode or deployed bytecode. The decompiler will automatically detect each of these cases. A usage example follows.

```
$ python2.7 main.py tests/bytecode/smallexample.bc
Successfully decompiled tests/bytecode/smallexample.bc
Running time: 0.178121
contract Decompiled {
    ...
}
```

Additionally, the `test` argument can be given to automatically run all the correctness tests. The test contracts are included in the `source/test/` directory.