VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Compiling Whiley for the Ethereum Virtual Machine

Dylan Kumar

Supervisors: David Pearce, Jens Dietrich

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

Ethereum is a blockchain based platform that supports a Turing complete contract language. However, methods of writing smart contracts have been error prone. This has resulted in many historically expensive bugs such as the DAO. Whiley is a programming language which allows you to verify code. This project looks at compiling Whiley to Ethereum bytecode in order to utilise Whiley's verification tools to make writing smart contracts safer. Finally, we run some coverage tests to see how the developed method adheres to the Whiley language.

Acknowledgments

I would like to give my thanks to my supervisor David Pearce for his invaluable teachings, advice and patience through this project. I am grateful to have the opportunity to work with him and make what I originally perceived as quite a daunting task, quite enjoyable and achievable. I would also like to give my regards to my co-supervisor Jens Dietrich for facilitating interesting discussion during meetings and providing another avenue for thought and ideas.

Contents

Figures

Chapter 1

Introduction

In recent years Cryptocurrencies have seen widespread adoption particularly with the rise of Bitcoin. At its peak, Bitcoin had a market capitalization of over 300 billion dollars USD [7]. Bitcoin [24] solved the problem of distributed consensus with the idea of proof of work, where updates in the state of the application would be collectively agreed upon. This idea is an integral one to a technology called the Blockchain which is the foundation of Bitcoin and many other Cryptocurrencies.

A Blockchain is a chain of records, where each record is named a block, and each block is connected to a previous block through a hash. The Blockchain takes the idea of a Merkle Patricia (hash) tree [39] and utilises a peer to peer approach in deciding whether a change to the tree is valid or not. This means the Blockchain allows for a decentralized system whereby no single entity is able to decide who controls the network [24].

Bitcoin has a primitive smart contract and scripting language, which allows code to be written and executed on the Blockchain [2]. Ethereum [39] is a decentralized blockchain-based platform that makes improvements on its predecessors such as Bitcoin. One way it does this is through providing a Turning complete Virtual Machine, with persistent state and improved storage efficiencies over its predecessor. This allows code to be written and deployed as ”smart contracts”. Smart contracts are a type of contract that are enforced by code and are executed upon meeting certain conditions. Being a code based irreversible contract, smart contracts do not require a third party to ensure credibility, and therefore remove the service fees and obligation to a middle man, as well as enable benefits in security. Popular examples of smart contracts include mortgages, insurance claims, payments and settlements, prediction or financial markets, or a Decentralized Autonomous Organization (DAO). A DAO is a decentralized organization that has rules enforced by its smart contract code. These rules are maintained in the Blockchain as well as financial records of the organization. “The DAO” was a DAO with the purpose of providing a crowdfunding platform for businesses and investors that utilized the Ethereum blockchain. The DAO had vulnerabilities in its smart contract, resulting in it being hacked and over 3.6 million ETH tokens being stolen in June 2016, which was worth over $70 million USD at the time [14]. The DAO showed the community the importance of code correctness in smart contracts, especially when dealing with large sums of money. Furthermore, as a contract cannot be “patched” or modified once deployed to the blockchain, any bugs that exist in a contract could have devastating effects. This resulted in a greater emphasis in producing bug-free code, with static program analysis techniques being created, such as MadMax, which automatically detects gas-focused vulnerabilities [17].

Popular Ethereum smart contract languages include Solidity, Vyper, LLL, and many more. These languages currently lack the tools that help to eliminate errors or vulnerabilities in smart contracts. Whiley [34] is a programming language designed to employ static

type checking to eliminate certain errors at compile time, such as divide-by-zero, array out-of-bounds and null dereference errors. Whiley makes use of pre, and post conditions that the programmer must specify to aid its automated theorem prover in detecting errors at compile time.

The code below illustrates a Whiley function, `abs()` which finds the absolute value of an integer `x` and returns it as a natural number, `r`. The function has two postconditions which are the two ensures clauses. The first postcondition checks that the return value, `r` is greater or equal to 0, and the second postcondition checks that it equals the value of `x` or negative `x`. For example, executing `abs(-4)` returns us the value `4` which is the absolute value of `-4`.

```
1 abs(int x) -> (int r)
2 // Must return natural numbers
3 ensures r >= 0
4 // Must return x or negative x
5 ensures r == x || r == -x:
6     if x >=0:
7         return x
8     else:
9         return -x
```

Listing 1.1: Whiley program for absolute value function

Whiley was first developed by David Pearce at Victoria University of Wellington and is an open-source project that has a small community of contributors. From catastrophes such as 'the DAO', we see there is a need to produce tools that assist in reducing bugs in smart contract code. The purpose of this project is to utilize Whiley's verification capabilities to explore ways to make Ethereum smart contracts safer. This project is going to move towards this goal by allowing us to compile Whiley programs into Ethereum bytecode. Compiling to EVM bytecode is one step in the bigger picture of making smart contracts safer by allowing us to explore the use of Whiley to find bugs in smart contracts.

1.1 Organisation

This report will begin by providing background information on the current state of Ethereum smart contracts. Through this, we will see existing solutions and how they tackle the problems Ethereum presents. We also contrast our solution with the existing solutions presented, and where the limitations of those may be remedied through our project. Through this contrast, we can also view the advantages of existing solutions over this project and potential limitations of them. We will then presents the work accomplished over this project, by first examining the architecture of Whiley and how our project fits within the current system, before delving into the details of the software implementation. Finally, we explore areas of future work, before concluding this report.

1.2 Contributions

- We have designed and implemented a translator from the Whiley programming language to Ethereum bytecode.
- We have performed an experimental evaluation using the existing Whiley test suite to identify which tests pass.

Chapter 2

Background

We begin by providing detailed definitions of key concepts related to the background of our work.

2.1 Introduction to Whiley

```
type nat is (int n) where n > 0 // Nominal
type ExposedSquare is {  int rank, bool holdsBomb }
type HiddenSquare is {  bool holdsBomb,   bool flagged }
type Square is ExposedSquare | HiddenSquare

// ExposedSquare constructor
function ExposedSquare(int rank, bool bomb) -> ExposedSquare:
    return { rank: rank, holdsBomb: bomb }

// HiddenSquare constructor
function HiddenSquare(bool bomb, bool flag) -> HiddenSquare:
    return { holdsBomb: bomb, flagged: flag }

type Board is {
    Square[] squares, // Array of squares making up the board
    int width, // Width of the game board (in squares)
    int height // Height of the game board (in squares)
}

function add(int[] v1, int[] v2) -> (int[] v3)
    requires |v1| == |v2|
    ensures |v1| == |v3|:
    int i=0
    while i < |v1|:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

Listing 2.1: Whiley program for Minesweeper game [31]

Whiley is a programming language that combines functional and imperative paradigms and allows for formal specification through preconditions, postconditions and loop invari-

ants [32]. Through these explicit specifications, Whiley is able to automatically reason about the validity of statements by making use of tools such as an SMT solver. Whiley uses indentation syntax over braces for statement blocks. However, although visually resembling a python-like syntax, the core of Whiley is functional and pure. Whiley has many data types, including integers, booleans, bytes, arrays, records, unions, nominals, references and functions. Listing 2.1 shows some of these datatypes in code. This code excerpt, taken from a Minesweeper game, illustrates the building blocks for the Minesweeper board.

Whiley programs are composed of functions and methods. The `function` keyword indicates a pure function, which refers to a function that always returns the same output given an input, as seen from lines 7 and 11. Methods are impure meaning an input may not always result in the same output and side effects may be observed, e.g. modification of input parameters or state outside the method.

A record opens with curly braces and ends with curly braces, and is comparable to a struct in C. Lines 2 and 3, show the record datatype which are composed of multiple fields. These records can be instantiated through their constructor as seen in lines 7 and 11. Line 4 shows us an example of a union type, which is a type that accepts any value held by its components. In this example, `Square` can either be an `ExposedSquare` or `HiddenSquare`. Line 15 shows the declaration of an array, and below it we see two integer datatypes. Nominal types are named types composed of an underlying type. Nominal types are often used to enforce certain rules or enforce information hiding. For example, on line 1, the nominal `nat` is an integer type that is required to be greater than 0 (natural number). This example also shows booleans (line 3).

The `add()` function on line 20 illustrates some of the Whiley verification focused language features. On line 21 the `requires` clause is used to impose a precondition on the function, whilst line 22 uses the `ensures` clause as a post condition. Through explicit specifications such as these clauses, asserts, assumes, and loop invariants, Whiley is able to reason about the validity of a program and its inputs. Before executing a Whiley program, the whiley file is first compiled into the Whiley Intermediate Language (WyIL) which represents the abstract syntax tree of the program. From this, the verifying compiler checks that the methods and functions meet the explicit specifications in the program, and reports any failures to the user.

2.2 The Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a virtual machine that can execute tasks to program some functionality on the Ethereum blockchain [39]. The EVM creates an environment that manages smart contract execution.

2.2.1 EVM Memory

Ethereum Bytecode is similar to Java Bytecode as both the Ethereum Virtual Machine (EVM) and Java Virtual Machine (JVM) utilize a stack machine. The Ethereum Virtual Machine is defined in the yellow paper [39], which defines the architecture and use cases of Ethereum. The Virtual Machine operates by pushing operations onto its Stack. Operations consist of an operation code (opcode) and operand (depending on whether the specific opcode requires it). The operation is the task to perform, and the operand is the value of the data input to allocated.

The EVM [40] has three types of memory, Stack, Memory, Storage. In the stack, new data is pushed onto it during execution. Memory is an expandable byte array and comparable to JVM's heap. Long term storage is a key value store that persists data. Both the stack

and memory (heap) are reset after computation ends with the EVM. The size of each item in EVM memory and storage, which is also called `wordsize` is 256 bits.

2.2.2 EVM Bytecode

EVM has 140 different opcodes which represent the instructions or specific tasks to be executed [18]. As classified by Hollander, the most commonly used opcode types can be split into the following categories for simplicity [18]:

- Stack-manipulating opcodes: (`POP`, `PUSH`, `DUP`, `SWAP`)
- Arithmetic/comparison/bitwise opcodes: (`ADD`, `SUB`, `GT`, `LT`, `AND`, `OR`)
- Environmental opcodes: (`CALLER`, `CALLVALUE`, `NUMBER`)
- Memory-manipulating opcodes (`MLOAD`, `MSTORE`, `MSTORE8`, `MSIZE`)
- Storage-manipulating: opcodes (`SLOAD`, `SSTORE`)
- Program counter related opcodes: (`JUMP`, `JUMPI`, `PC`, `JUMPDEST`)
- Halting opcodes: (`STOP`, `RETURN`, `REVERT`, `INVALID`, `SELFDESTRUCT`)

Opcodes are encoded to bytecode for efficient storage. Bytecodes are stored in hexidemical format. For example, `PUSH1` is `0x60`, or the integer 96.

The following bytecode shows the integer `6` being subtracted from the integer `9`. EVM executes this by pushing on the second value (`6`), pushing on the first value (`9`), and then executing the arithmetic operator which in this case is `SUB`, which pushes the result.

```
1 PUSH1 0x06
2 PUSH1 0x09
3 SUB
```

Listing 2.2: Subtracting two numbers in EVM bytecode

2.2.3 Gas in Ethereum

There are two types of tokens in Ethereum: ether, and gas [39]. Ether is a cryptocurrency that holds some intrinsic value based off of the supply and demand of the token. Ether can be used to purchase gas. Having gas as a separate resource to Ether was done for a variety of reasons, e.g. allowing Ether to fluctuate in price and not affect the price of gas, and vice versa [6].

Ethereum uses the concept of gas as a commodity or resource that allows operations to be executed [9, 6]. The purpose of gas is to constrain what can be executed on the EVM to decentivize overwhelming the network and executing costly transactions. This helps to protect against Denial of Service attacks (DoS). Furthermore, by requiring a cost to execute transactions Ethereum incentives minimising the number of instructions in a contract.

During execution, Ethereum contracts only run as long as there is sufficient gas to execute contract code. An out of gas operation is when a contract runs out of gas during execution and therefore the Ethereum Virtual Machine throws an out of gas execution and aborts contract execution. If the contract runs out of gas during execution, and the transaction is terminated, the caller loses all provided gas. On the other hand, the successful termination of a transaction results in any excess/remaining gas being returned to the caller [9].

2.3 Introduction to Solidity

Solidity is an object-oriented, high level programming language for writing smart contracts that compiles to Ethereum Bytecode. Solidity is the most popular programming language for implementing smart contracts on Ethereum. Solidity utilizes a JavaScript-like syntax, as we see below from the following example smart contract [16].

```
1  pragma solidity ^0.4.22;
2  contract SimpleBank {
3      uint8 private clientCount;
4      mapping (address => uint) private balances;
5      address public owner;
6
7      constructor() public payable {
8          require(msg.value == 30 ether, "30 ether initial
               funding required");
9          /* Set the owner to the creator of this contract */
10         owner = msg.sender;
11         clientCount = 0;
12     }
13
14     /// @notice Enroll a customer with the bank,
15     /// giving the first 3 of them 10 ether as reward
16     /// @return The balance of the user after enrolling
17     function enroll() public returns (uint) {
18         if (clientCount < 3) {
19             clientCount++;
20             balances[msg.sender] = 10 ether;
21         }
22         return balances[msg.sender];
23     }
24
25     /// @notice Deposit ether into bank
26     /// @return The balance of the user after the deposit is
           made
27     function deposit() public payable returns (uint) {
28         balances[msg.sender] += msg.value;
29         emit LogDepositMade(msg.sender, msg.value);
30         return balances[msg.sender];
31     }
32 }
```

Listing 2.3: Simple banking contract in Solidity

This smart contract acts as a simple banking system, where customers can enroll and deposit ether into the bank. As seen in line 17, the `public` identifier sets the visibility of the function `enroll()` to public. This means that anybody who has access to `SimpleBank` on the Blockchain that contract has been deployed to can invoke this function. The constructor, on line 7, is called whenever the smart contract is deployed to the blockchain. In the constructor we require 30 ether to be funded to the contract for it to be deployed, and following that set the owner of the bank to the sender of those funds, and set `clientCount` to `0`. The keyword `payable` as seen on line 27 and 7 is required to allow ether to be received by the

contract. The `returns (uint)` on `enroll()` and `deposit` tell Solidity that an unsigned-integer (`uint`) needs to be returned from these functions. `balances` provides the mapping of user addresses to the amount of ether stored in the account. There are many special variables and functions which exist in the global name space such as `msg.value`, `msg.sender` and `block.number` [1]. `msg.sender` as seen on line 5 is one of these implicit parameters that indicates who called this contract. `msg.value` on line 28 provides us with the "number of wei sent with the message"[1] (wei is a denomination of ether). Overall, `SimpleBank` is a contract where users can enroll and deposit either.

2.4 Attacks that can be executed on Ethereum smart contracts

Atzei, Bartoletti, Cimoli and others discuss a variety of different ways Ethereum smart contracts can be attacked [9, 38]. I am now going to discuss some of these in more detail.

2.4.1 Unchecked Send

The send function can be used to send Ether to an address [38]. However, the send function can fail. If this happens, the code following the invocation will still be executed if the return of the send function is unchecked. For example, consider the following illustration where `prizePaidOut` will be set to `true` whether or not `winner.send(1000)` succeeds.

```
1 if (gameHasEnded && !(prizePaidOut)) {
2   winner.send(1000); // send a prize to the winner
3   prizePaidOut = True;
4 }
```

Listing 2.4: Unchecked send in Solidity contract [38]

The `send()` function may fail if `winner` is not correctly defined, e.g. if `winner` is a contract that throws an exception resulting in the send failing [38]. However, the `send()` may also fail even if `winner` is correctly defined. EVM manages its operations through a callstack which is limited to a max depth of 1024 [39] so the `send()` function may fail if EVM is at the callstack limit. This is called a 'callstack attack'. 'BTC Relay', 'King of the Ether Throne', and etherpot are all examples of famous contracts that have exhibited this bug [23, 36, 22]. As Solidity limits gas usage to 2300 for a `send` function, the `send` may also fail given a computationally intensive fallback function, though this was later patched as will be discussed in the following section.

2.4.2 Call to the Unknown

As EVM bytecode does not support the concept of functions, Solidity uses a simple function dispatching mechanism where each function is identified by a signature. If a contract is called and the function signature is matched, the EVM jumps to the area of the functions code, but if no signature is matched the EVM jumps to the fallback function. The fallback function is a function without name/arguments and does not return anything [37], as seen in Listing 2.5.

In Solidity there may only be one callback function. This fallback function is represented by `function()`, as seen in the example above. If the address at the function signature that is called does not exist, or incorrect parameters are supplied, the fallback function is invoked instead. This means that the fallback function can be unintentionally called given the incorrect invocation of a function. This may be as simple as specifying the wrong parameter type, or failing to provide a parameter.

```
1  contract Fallback {
2      function() payable {
3          // empty fallback function
4      }
5  }
```

Listing 2.5: Fallback function in Solidity

'The DAO' attack, explained simply, was caused using repeated invocations to the contracts fallback function [9]. Another popular case, 'King of Ether Throne' was also attacked through the fallback function to steal ether, in combination with other vulnerabilities. We will not go into detail in how the fallback function can be attacked. These attacks can be examined in more detail in the paper [9].

Subsequently, Solidity made changes from v4.0.0 [37, 26], which now require the `payable` modifier to be supplied in order for the fallback function to be able to accept ether. The example above uses the `payable` modifier. It's exclusion means that the fallback function cannot accept ether transfer. Although this helps minimise the risk of losing money given the invocation of a wrong function in a contract, it does not mitigate the risk of attaching ether to a transaction on a wrong contract that has a payable fallback function. Updates were also made so that if the fallback function was not explicitly defined, the default fallback function would `throw` which returns an error, and reverts state changes. This helps to prevent ether being incorrectly transferred to a contracts account where the contract does not have a payable fallback function explicitly defined.

Solidity also limited the available gas to the fallback function to 2300, which is only enough to do basic logging operations [37]. The purpose of this was to stop contract makers from implementing business logic in their fallback function, as the fallback function can be invoked by anyone that has access to the contract.

2.4.3 Reentrancy

EVM does not allow concurrent function executions. In EVM, when a contract calls another, the current contract waits for the caller to finish executing before proceeding. A reentrancy attack can be executed by repeatably calling a single function before the invocation of that function had finished [9, 21]. Because the current execution will wait for the caller to finish executing before continuing, a contract's state can be partially altered at some point without finishing execution. This can cause unintended consequences. For example, consider Listing 2.6.

In an attempt to send funds to the sender (withdraw funds), `msg.sender.call()` is used to call the fallback function on the contract referenced by `msg.sender`. On line 6, the callee contract `Attacker` can call withdrawBalance() again using its fallback function resulting in balance being withdrawn again and again. As the users balance is only set to 0 after the invocation of the function, repeated invocations prior to this would succeed. Although not obvious, a solution would be to set the users balance to 0 before the invocation of the call, hence preventing repeated invocations from succeeding in withdrawing funds. Cross-function reentrancy attacks are also possible, where a similar thing is done but with two functions and/or contracts that share the same global state. 'The DAO' attack mentioned prior, is an example of an attack that used both reentrancy and cross-function reentrancy [21].

```
1  contract Fund {
2      mapping (address => uint) private userBalances;
3      function withdrawBalance() public {
4          uint amountToWithdraw = userBalances[msg.sender];
5          (bool success, ) = msg.sender.call.value(
               amountToWithdraw)(); // At this point, the caller
               's code is executed, and can call withdrawBalance
                again
6          require(success);
7          userBalances[msg.sender] = 0;
8      }
9  }
10 contract Attacker {
11     Fund f;
12     // Constructor
13     function Attacker(address fund) payable {
14         f = Fund(fund);
15     }
16     // Fallback Function
17     function () payable {
18         f.withdrawBalance();
19     }
20 }
```

Listing 2.6: Reentrancy Attack through repeated calls [5]

2.4.4 Integer overflow/underflow

Integers on the EVM are bounded to a fixed size [20]. For example, an 8 bit integer (`uint8` in Solidity) would represent the numbers from 0 to 255. This means that storing 256 into a `uint8` would store the value 0. An underflow is where the value flows below the lower bounds [10]. For example, subtracting 1 from a `uint8` that holds the value 0 will give us the value 255. An overflow is where the value flows above the upper bounds [10]. For example, adding 1 to a `uint8` that holds the value 255 or assigning a value above the upper bounds.

Integer over/under flows cause the value to wrap around. This can be problematic if the programmer is not paying careful attention and exceed the bounds of an integer in some way, perhaps by adding two large numbers together for instance.

A popular case of this is Proof of Weak Hands Coin (PoWHC), which was a satire contract devised as a Ponzi scheme by users on the website 4chan [11]. It's value grew to close to a thousand Ether within days, before being exploited and 886 Ether being transferred out of the contract [11].

PoWHC used an implementation of ERC-20 as their coin, which is a popular token standard for Ethereum [11]. However, their implementation contained a vulnerability. Their system could allow for users to approve selected users to transfer tokens on their behalf. This meant that another user could sell tokens for that user. However, there was a bug in the contract meaning that selling account A's tokens from account B would result in the sold token amount being subtracted from the account B's balance, not A. If the balance resulted in an integer underflow, account B would be left with an extremely large balance of tokens. This can be seen in Listing 2.7. On line 5 `msg.sender` is assumed to be the seller, instead of the actual seller being passed into the `sell` function from the method that calls it. Therefore,

`msg.sender` refers to the current account, B, instead of A.

```
function sell(uint256 amount) internal {
    var numEthers = getEtherForTokens(amount);
    // remove tokens
    totalSupply -= amount;
    balanceOfOld[msg.sender] -= amount;

    ...
}
```

Listing 2.7: PoWHC Vulnerability [11]

Whilst Overflow checking could be done at runtime it would be quite expensive for the EVM, and as gas is used for operations, it is important to be as computationally efficient as possible. This is one reason for why most Solidity smart contracts are 200 lines or less [19]. However, in Soldity 0.4.16, the Solidity Compiler, "now include experimental support for automated overflow and assertion checking at compile-time using the SMT solver Z3" [13], which helps to mitigate this problem.

2.5 Existing Solutions

Following the DAO incident, many attempts have been made to reduce the amount of bugs in writing smart contracts, and a variety of verification tools have been produced. The tools either perform verification prior to contract deployment, or after, with the majority providing post analysis at the bytecode level. This background Survey will present existing solutions, and look at how they verify smart contracts.

2.6 MadMax

MadMax is a static program analysis technique that helps to identify gas-focused vulnerabilities in Ethereum [17]. MadMax uses a decompiler to provide static program analysis on EVM bytecode by taking EVM bytecode and converting it to structured intermediate language. MadMax identifies a variety of vulnerabilities that result in out of gas exceptions, such as unbounded mass operations, unbounded externally controlled data structures, non external isolated calls, and integer overflows.

Performing static analysis on EVM bytecode is challenging because EVM bytecode has many differences to other bytecode languages such as JVM. For example, lacking the concept of functions, differences with jump positions and destinations and control flow edges [17]. Some of the challenges with decompiling EVM bytecode include [17]:

- JVM has clearly defined jump targets independent from the stack, whilst EVM jump destinations are variables read from the stack, and depend on the order of stack contents.
- JVM uses defined jump destinations as above point, for method invocation and return instructions. In Ethereum bytecode there is no concept of a function, instead EVM makes use of unstructured control flow that requires you to branch to a particular destination in code when that operation is reached.
- EVM bytecode does not have the concept of structures or objects, unlike JVM bytecode.

Due to these challenges, the MadMax decompiler performs value-flow analysis and detects function boundaries with a high-fidelity analysis in order to produce control flow graphs [17]. From these CFGs, analysis can be performed on the semantics, memory layout, and other areas that may result in vulnerabilities.

Unlike with MadMax and other tools we will discuss, our project does not face the difficulties of flow-analysis, as Whiley provides static analysis on the Whiley language level. Therefore, analysis is not done on or during the conversion from Whiley to EVM bytecode. Whether this could be in fact limiting could be a topic for future research.

Maxmax was tested by validating all 6.3 million contracts deployed on the Ethereum blockchain. With a reported precision of 80% based on a manual inspection of samples, "The analysis reports vulnerabilities for contracts holding a total value of over $2.8B" [17]. A positive of MadMax's analysis techniques are that it provides analysis on EVM bytecode hence contracts can be written in any language that compiles to EVM and be analysed. This means MadMax does not require contract source code, so bytecode analysis can be applied on new and prior contracts deployed on a blockchain. Furthermore, it means that using MadMax, EVM smart contracts written in Whiley could be validated for vulnerabilities.

2.7 Formal Verification of Smart Contracts

'Formal Verification of Smart Contracts' [12] discuses a framework that can be used for the analysis and verification of the runtime safety and functional correctness of Ethereum Smart Contracts. This was achieved in a similar way to MadMax, by making use of a decompiler to translate contracts to a functional programming language called F*. The majority of contracts on the Ethereum blockchain do not have publicly available source code. "At the moment of this writing, only 396 out of 112,802 contracts have their source code available on http://etherscan.io" [12]. This means that analysing lower-level EVM bytecode is necessary to get a full picture of the vulnerabilities that most smart contracts have on the Ethereum Blockchain.

The paper presents two tools for converting contract code to F* for verification. Solidity* and EVM*. Solidity* converts Solidity to F* which allows the source level verification of "functional correctness specifications (such as contract invariants) and safety with respect to runtime errors" [12]. EVM* converts EVM bytecode to F* which allows for analysis of "low-level properties, such as bounds on the amount of gas required to complete a call or a transaction" [12]. This is partly done by evaluating jump destinations and detecting stack under or overflows. Furthermore, as EVM uses `JUMPDEST` to identify a jump destination, any jumps to invalid jump destinations can be highlighted. The F* language generates automated queries for an SMT solver which statically verifies pertinent properties of a smart contract. F* can be used to detect vulnerabilities such as reentrancy type errors, as the tool checks vulnerable patterns such as not checking the result of external calls.

2.8 Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL

In this paper, Samani, Bortin, et al. discuss their work extending an existing EVM formal model in Isabelle/HOL [8]. Isabelle/HOL is an interactive and higher order logic theorem prover that uses handwritten theorems composed pre and post conditions to verify program correctness. The extension presented is an attempt to verify smart contract based on correctness properties and support automated generation of verification conditions. Isabelle/HOL,

like other contract verification tools such as MadMax and F*, also chose to target unstructured bytecode for similar reasons that we've discussed. However, unlike MadMax which decompiles to an intermediate language, the authors argue that a more general approach means that there is less reliance on the correctness of "higher-level tools", such as the translation to one formalism to another.

The purpose of this project is to set the ground work for "full functional correctness of Ethereum smart contracts" [8]. The tool verifies 36 commonly used instructions (compared to 70+ instructions that EVM provides). Furthermore, the project has been "accepted in the official EVM formalisation repository maintained by the Ethereum foundation" [8].

Isabelle/HOL provides decompilation through extraction of Control Flow Graphs (CFG) by splitting a program into 'basic blocks' which are a sequences of instructions without jumps. Based on the basic blocks, the authors created a series of rules that specify the correct behaviour of basic blocks and the connections between them, taking into account properties such as the program counter, stack height, gas, and other properties.

To verify a contract, a specification of theorems tailored to that contract must be written. This requires making statements about properties of the EVM program at a low level, which involves referring to low level properties such as where the address is located in storage. Because these theorem's are written for contracts after a contract is written/deployed, the pre/post conditions can be more difficult to read than in-code specifications in Whiley and require manual identification of variables in contract storage as opposed to referencing a variable name. So whilst the use of pre/post conditions appears similar to be similar to how Whiley can verify contracts, this tool appears to be complicated. Whether this provides better contract correctness than Whiley however is a question that could be later explored.

2.9 Oyente

The Oyente verification tool is based on "symbolic execution" [21] which refers to representing the different paths of the program and statically reasoning about it to work out which paths are feasible and infeasible. It does this generating a control flow graph based on the EVM bytecode. The control flow graph is then passed into the symbolic explorer which explores all the possible paths of the program. Every path that can be explored is passed into the core analysis which detects whether that part of the program has some potential vulnerability. If there is, this result is forwarded to the validator which validates whether the result is positive. The control flow graph is used to visit every execution trace of the program. To ascertain whether an input satisfies a certain condition, a theorem prover (Z3 Bit-Vector Solver) is passed the equation of the condition which returns whether such an input exists.

Two unique contributions the authors made at the time were the detection of the timestamp dependence and transaction ordering dependence vulnerabilities.

2.10 ZEUS: Analyzing Safety of Smart Contracts

ZEUS is another verification tool [20]. There are 4 primary stages in ZEUS's verification pipeline, as see in Figure 2.1. Firstly, a user must write out a set of criteria which will act as pre/post conditions for the contract. This is achieved through a file with a XML style format/syntax that describes the conditions/predicates where the asserts should be placed. Next, the contract is converted into an intermediate representation (IR), such as LLVM bitcode where the policy specifications are inserted into the contract bitcode as asserts. The points at which the asserts/predicates are placed are determined through static analysis on

Figure 2.1: The pipeline of ZEUS [19]

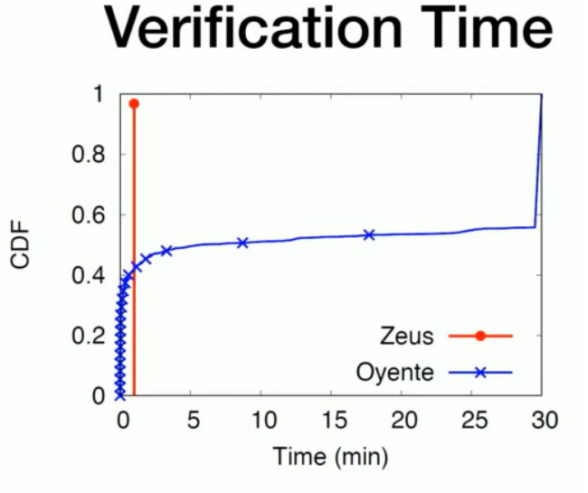

Figure 2.2: Zeus vs Oyente Verification Time [19]

the IR. The modified IR is then fed into the verification engine which uses "constrained horn clauses (CHCs) to quickly ascertain the safety of the smart contract" [20]. Another advantage of using LLVM bitcode is that any verifier that works with LLVM bitcode can be used.

The authors discuss several motivations for creating ZEUS. They mention the limitations of Oyente, including its bug detection at bytecode level being neither sound or complete. F* on the other hand leaves out constructs such as loops. "93% of contracts contain loops" [20], so this is an important aspect to consider. Finally F* requires manual proofs, which have to be written out per contract; something people may not want to do.

Zeus is considerably faster at verification than Oytene, as can be seen in Figure 2.2. ZEUS boasts zero false negatives due to conservative taint analysis, with far less false positives than Oyente [20]. Furthermore, ZEUS is more scalable due to less verification overhead and quicker speeds. For example, the `send` construct can no longer cause reentrancy due to send limiting the fallback function to 2300 gas. This is not enough gas for storage or function calls. However, Oyente examines `CALL` bytecodes when checking for reentrancy, which both `send` and `call` map to. The authors of ZEUS argue that this is the cause of the false positives that Oyente generates. Examining `CALL`'s further for their differences is one improvement ZEUS has made to reduce false positives in reentrancy vulnerability detection.

Chapter 3

Design

We will now give an overview of the design behind our Ethereum translator, in particular focusing on the layout of memory through a callstack and heap.

3.1 Building upon Existing Architecture

The current architecture of the Whiley compilation process involves Whiley files being converted through the language compiler (WyC) into Wyil files. Wyil files are an intermediate language.

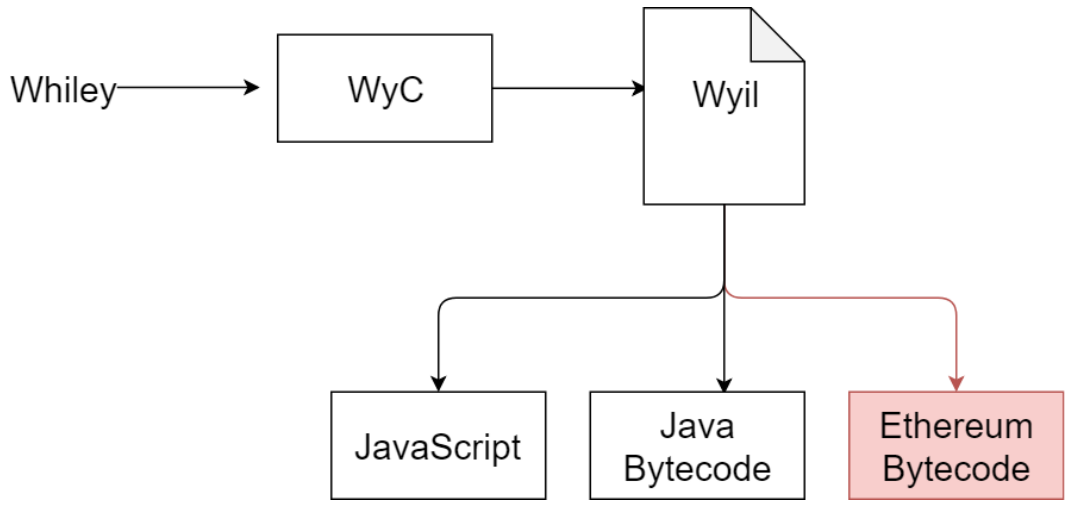

Figure 3.1: Whiley Compilation Process

Whiley currently compiles to JavaScript and Java Bytecode, and attempts have been made to compile to other languages such as C. The overall architecture can be seen in Figure 3.1. Our Ethereum compiler visits all parts of the Abstract Syntax Tree of the Whiley Intermediate Language file, and translates them into Ethereum bytecode. As typechecking is already completed by the Whiley language Compiler, my implementation has been purely focused on the translation to bytecode, rather than ensuring the validity of a particular program.

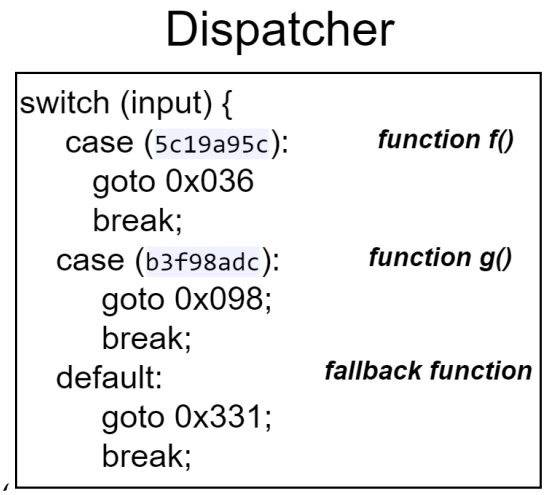

Figure 3.2: Example dispatcher code

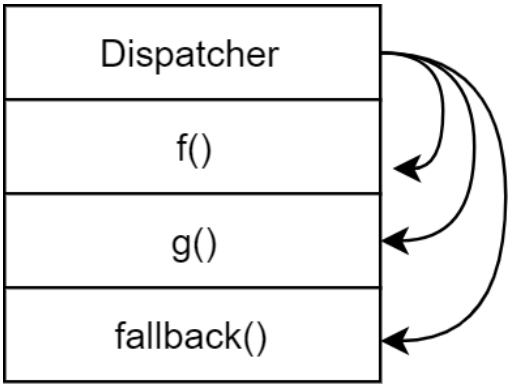

Figure 3.3: Dispatcher

3.1.1 Overview of Contract Execution

To call contract code, there needs to be some way of interacting with the deployed contract which is stored as bytecode on the blockchain. An Application Binary interface (ABI) is the common approach used to provide the information needed to call a contract. An ABI is an interface that allows you to encode contract calls for the EVM or decode data out of transactions. An ABI tells us the parameters required to invoke a specific function when calling a contract, and provides the return data in an expected format. The Solidity ABI is a popular example of an ABI for the EVM [3]. Functions in Solidity contracts are identified through a unique hash based on their function identifier. The Solidity ABI can be used to hash the identifier of the function we wish to invoke, so we can dispatch to this function in bytecode.

A dispatcher allows you to jump to a specific function. The Solidity dispatcher takes the the hash of the function identifier in a switch-like block and jumps to the associated position of that function in bytecode. Figure 3.2 shows some example code that provides an idea of what this could look like, where if the input matches any of the hashes, we jump to that function, otherwise invoke the fallback function. These jumps are visualised in Figure 3.3.

Our Compiler made use of an in compiler-ABI which records function information, and a primitive dispatcher that jumps to a compile time specified function. The implementation of this will be discussed in Section 4.1.

3.2 Memory Layout

The Ethereum virtual machine provides no mechanism for allocating space for variables or other data structures that we need. Therefore, we have to implement this ourselves. Storing into memory is important as using EVM stack memory would not allow us to access any particular element when we need it. Furthermore, Whiley has dynamically sized data structures such as unlimited sized arrays and open records. This means there needs to be some way of storing these such as in a heap. EVM memory was partitioned into two sections. The callstack, and the heap. This layout can be seen in Figure 3.4.

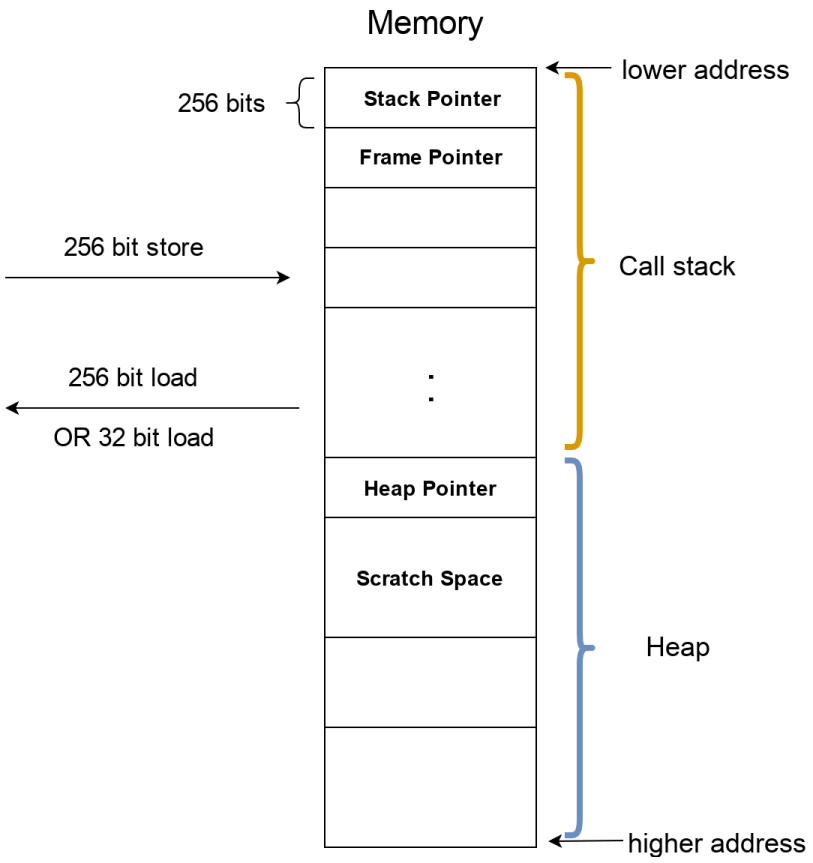

Figure 3.4: Memory in Ethereum Virtual Machine

```
// main function
function P1():
    int LV1 = 1
    P2(11, 5)

function P2(int p1, int p2):
    int LV1 = 7
    bool LV2 = false
    P3()

function P3():
    bool LV1 = true
```

Listing 3.1: Three functions in Whiley code

3.2.1 Call Stack

A callstack is a typical approach to representing the state of a machine code program. A callstack allows for keeping track of the state and variables of multiple different functions in memory, as they are invoked during the lifetime of a program [35]. The callstack is comprised of stackframes, each of which represent a function call and its arguments. When a function is invoked, a new frame is pushed onto the stack which contains the function return address, arguments and local variables.

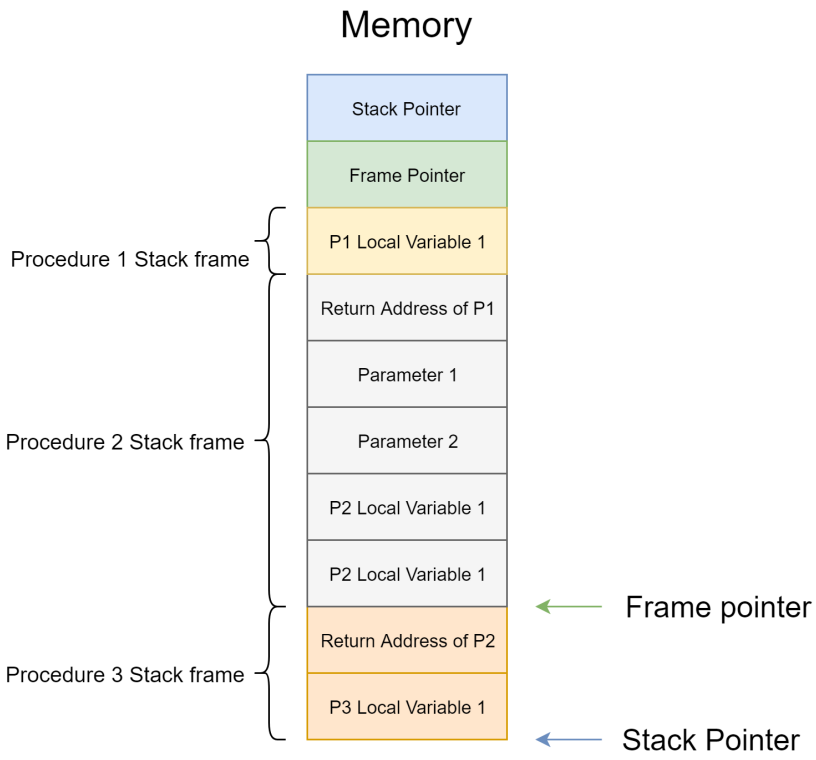

Figure 3.5: The Callstack in Memory

It is common on the EVM to use the concept of a free memory pointer to maintain a record of the position that the next byte of memory can be allocated to. Memory in EVM can be allocated through the operation `MSTORE` and loaded through `MLOAD`. Our system uses a free memory pointer called the stack pointer to maintain this position, which is a component of the call stack. A callstack makes use of a stack pointer, and frame pointer. The stack pointer stores an address that points to the top of the stack. This position is where the next stackframe will be allocated to, and is the end of the active (highest) stack frame. The frame pointer, is used to point to the start of the active frame in the stack.

Within the scope of a particular function in the compiler, variable names are associated with an offset. This offset indicates how far along the active stackframe the variable is stored at. Therefore, the frame pointer added to the offset will provide the variable location in memory. Each function is abstracted in my translator using a `FunctionScope` object. Hence duplicate variable names within different functions of a program are uniquely associated with the function scope they belongs to, meaning variables of the same name within different functions will not point to the same position in memory.

For instance, consider the Whiley code in Listing 3.1. In this code we have 3 functions, each of which are visualised as procedures in the callstack in Figure 3.5, where *Procedure 1* represents `P1`, *Procedure 2* represents `P2()`, and so on. Notice that in both `P1()`, `P2()` and `P3()` there is a variable named `LV1`. However, `LV1` is of different types and values in each function, and are stored in their associated stackframe in the callstack as seen from their associated procedure. Despite having the same variable name, their position in memory is distinct, and these positions are maintained in the `FunctionScope` object which holds their offset in memory in their stackframe. The procedure also maintains the return address of the previous procedure so after that function has been invoked the EVM can jump to where that function was called before proceeding. Notice that `P1()` doesn't have this return address however, as it is the first function called (main function). Also notice that, the function `P2()`

takes in two parameters which are held in *Procedure 2* after the return address of `P1()`.

After the invocation of a function, the frame pointer is set to the value of the stack pointer, and the stack pointer is incremented based on the amount of memory it needs to allocate. Likewise, after the function has been invoked, the frame is popped off the stack, which results in the stack pointer being set to the value of the frame pointer, and the frame pointer is moved down to the start of the previous frame.

3.2.2 Heap

Whiley has dynamically sized data structures, such as arrays and records, so a heap was used for the allocation of these. The heap is a region in EVM memory set aside for the allocation of arrays and records, as well as scratch space. The heap begins at a user-defined index in EVM memory after the callstack, which allows for the callstack to be expanded or contracted depending on where the heap begins. This was set to word 25 in EVM memory, which is sufficient for the test cases. If a larger callstack was needed, this value can be changed easily. The heap utilises a heap pointer which maintains the position for the next free position in heap memory. After the heap pointer, another 10 words are reserved as scratch space for various operations. The use of the heap will be discussed in Section 3.3.

Using a separate section of memory for dynamically sized data structures (composite types), means that these structures can be resized/reallocated without impacting the variables within a stackframe. Although composite types could be stored on the callstack, because they can be of arbitrary length this could be difficult to achieve as you would have to copy the whole composite structure as the EVM exits from one function into another function. Instead, a pointer to the section of memory in the heap can be maintained, and this pointer can be copied. Furthermore, as the stackframe holds references to composite structures stored in the heap, a reallocation of one of these structures would mean the current stackframe would only need to update its reference to point to the newly allocated structure.

3.3 Composite Data Structures

Arrays and records (composite data structures) are both stored in the heap using a uniform representation for simplicity. Creating a composite data structure results in memory allocation from the next available slot (heap pointer value). The first (256 bit) word of a composite data structure holds its length (number of values). For each value of the composite structure, the next word is used to hold a value, followed by a word that indicates the type of value. This second word indicates the type of the value, primitive types hold 0, whilst pointer types store 1. Figure 3.6 shows an array that is allocated on the heap, where the first word holds the length of the array, and each following holds a value and type of value respectively. As the array holds 3 values (integers 1, 2 and 3) which are all primitives, the type of each is indicated as primitive with 0 at each type word.

Each value is stored in a word for simplicity, as storing multiple values in one word can be difficult in EVM and result in more complex operations. This is not an optimal solution, and is inefficient because 255 extra bits are being utilised within each type word (1 bit is sufficient to represent two types). It was done this way so that the uniform representation of composite structures could be iterated over in the same way, so only one one equals and clone method had to be written. Using 256 bits for each type tag is huge amount of unneeded space as we are only encoding 1 bit of information (1 or 0). Encoding the types within composites could have been restructured in a number of ways in order to use less memory. For example, given a composite with length of 256 elements or smaller, all the tags

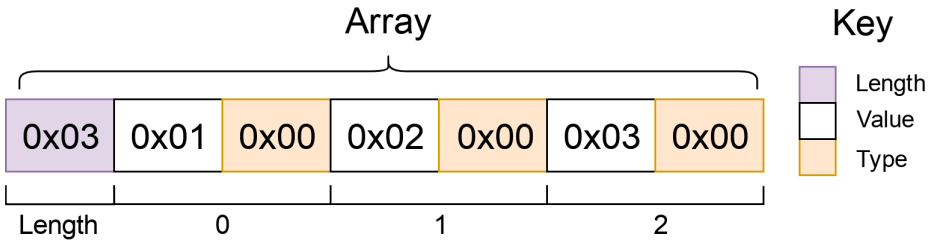

Figure 3.6: An Array on the heap

```
1    {int a, int[] b} rec = {a: 9, b: [7]}
```

Listing 3.2: A Whiley record

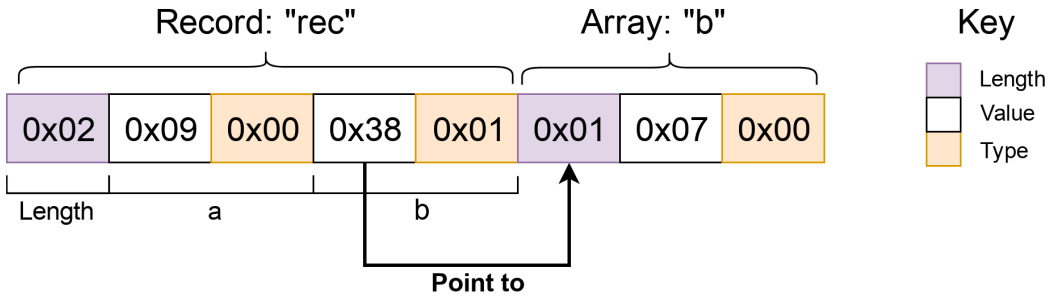

Figure 3.7: Composite data structures on the heap

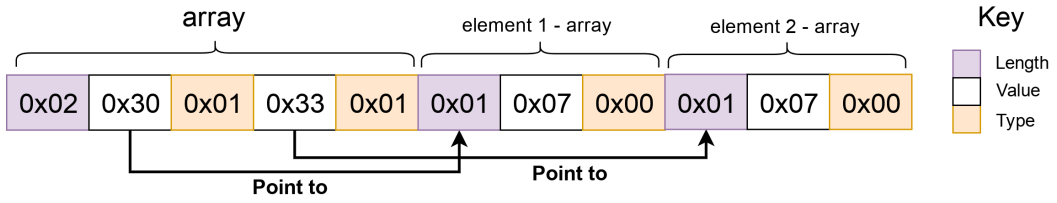

Figure 3.8: A 2D array on the heap

for could be compressed into one word. Each bit of the word would represent a type within the composite (pointer type or primitive). If more types were added, this could be extended to use two bits or more to encode each type. With this design, a composite with 256 elements would use 256 words for each value, with one word to encode the types, and one word to hold its length.

Whilst my design of composites is not optimal, it does facilitate the future extension of the compiler to support union types. Union types allow multiple types to be stored in a data type. For instance, an array that can store both integers and booleans. Keeping the structure of my design the same, union types could work by using more tag values for the new types that unions would allow (instead of just 0 and 1, could have tag values 2, 3, 4... for however many types the union would support). Although inefficient, our structure provides us the convenience of being able to extend the compiler under less difficulty.

A composite data structure can be made to hold another composite data structure. For example, an array that stores arrays (2D arrays), or a record that stores an array or record. Consider the Whiley code in Listing 3.2 which shows both a integer, and an array of integers inside a record.

Figure 3.7 shows the representation of this record in memory. The field `a` directly places the value `0x09` in memory, whilst field `b` stores a pointer to the array. The word holding `0x01` after the memory address `0x38` indicates that this is a pointer type. Similarly, Figure 3.8 shows a 2D array where each element of the top most array points to the beginning of the associated sub array. Note that self referential arrays are prevented in Whiley due to value semantics so cycles cannot occur.

Chapter 4

Implementation

In this chapter we are going to discuss the implementation of some of the features of our compiler, such as variables, patching, short circuit evaluation, control flow and composite data structures. I am now going to walk you through some of the details.

4.0.1 JEVM

This project utilizes a Java implementation of the Ethereum Virtual Machine called JEVM [29]. JEVM is a library that contains definitions and implementations of the various operations in EVM. For instance, in the bytecode class in JEVM static variables are associated with each EVM operation. Using JEVM is useful as it abstracts out the EVM operations and allows us to refer to them by name instead of bytecode number. JEVM provides us with a nice API that allows us to manipulate bytecodes and makes it easy to execute.

However, there are limitations to using JEVM. Firstly, it is still being developed, and not every Ethereum operation has been implemented, and the ones that have been implemented are not necessarily correct. We found bugs in the implementation of `SUB` and the comparison operators for example, which have since been fixed. As JEVM is implemented by my supervisor David Pearce, bugs found can be patched in-house, and any questions regarding its implementation can be answered easily. Although convenient, there are a number of other options we may want to consider in the future for an EVM implementation. For example, the Java implementation EthereumJ [25]. Using other implementations may help us verify whether the bytecode produced by JEVM is correct, and to examine what differences may be noticed between executed bytecode from the EVM's, if any.

4.1 In-Compiler ABI and Dispatching

Our compiler makes use of an in-compiler ABI, and doesn't support the use of an existing ABI such as the Solidity ABI. Because there is no external way to interact with our contracts, Solidity contracts cannot call Whiley contracts. The in-compiler ABI works by recording function information such as function jump locations. A primitive dispatcher is used within the bytecode of a program, where we begin by dispatching to a specific start function decided upon at compile time. This simplification means that when a contract is executed, it will always execute in the same order from a main function. My compiler has been set to dispatch straight to the `test()` function, which is the name of the main function in the test suite, though can be set to the name of any function in the program. This means that unlike Solidity's ABI, which allows any function to be invoked in a deployed contract by jumping based on a specified hash, a contract created with my compiler will always dispatch to a

function specified at compile time. This also means that there is no currently no fallback function.

4.2 Code Generation

An `Environment` class was created which facilitates the layout of memory. `Environment` provides an abstraction layer that handles the setup, allocation and retrieval of memory when storing variables. It also maintains the locations of the memory pointers (stack, frame, heap pointers), heap start position, scratch space, and other low level details. This means that if we wanted to make our compiler backwards compatible with Solidity, changes could be made, e.g. in Solidity the free memory pointer is stored at the second memory index (`0x40`) instead of the first index.

4.2.1 Patching

EVM jump targets are variables read from the stack. When receiving an instruction that a block of code needs to be jumped over (skipped), the translator does not necessarily know ahead of time where the target is. Furthermore, it needs to work out where to put the jump destination, `JUMPDEST`, which acts as a target you can jump to. To solve this problem, a temporary jump target variable is added on the stack (`0x00`), and no jump destinations are added. We leave these to be later replaced with the correct values at the correct positions, as seen in Figure 4.1. We call this *patching*.

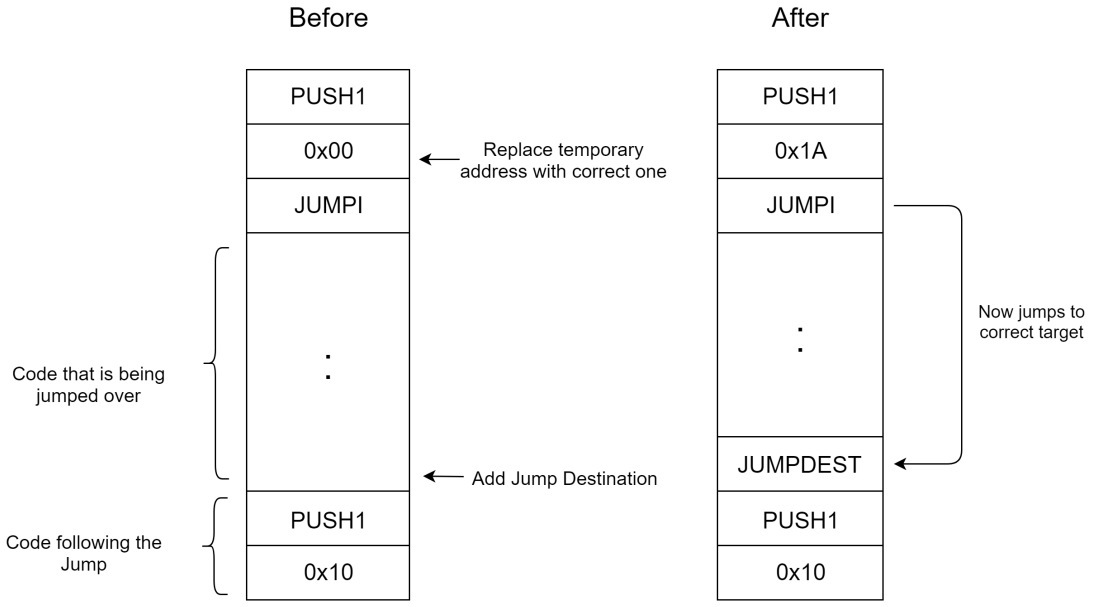

Figure 4.1: Patching a jump address to point to our target variable on the EVM stack. This target variable is added as a `JUMPDEST`

From Figure 4.1 we see the replacement of the incorrectly stored jump target on the stack, and a Jump Destination is added (`JUMPDEST`s) for the position our target should jump to. The temporary address is replaced with the target address and a jump destination is added at this target, resulting in the program being able to jump to the correct target. Because EVM reads static jump targets, adding a `JUMPDEST` means the target address and/or jump variable

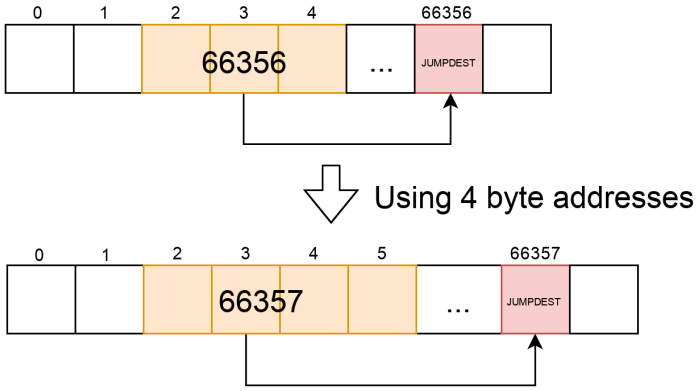

Figure 4.2: Branch offset for 3 byte addresses compared to 4 bytes. As we increase the size of the address, we need to move the location of our jump destination

location may now be incorrect. Hence, the affected patches had to be updated for an added `JUMPDEST`.

4.2.2 Jump Addresses

One assumption we made for this project was that all jump addresses would use a fixed amount of three bytes. This was to simplify patching, as increasing the number of bytes for a patch increases the distance between that patch and other patches. This will mean every patch between the added one will have to have its variable address position and jump destination updated to accommodate for the size of the added one. As these addresses are not of a fixed size more calculations/shifting will have to take place, resulting in a cascading effect where changing one patch may result in another patch to be changed which may affect other patches and so on. A simple example of this can be seen in Figure 4.2 which shows that using a larger address would result in the movement of the jump destination. In this figure we can see that as we increase the address variable from 3 bytes to 4 bytes, the position it jumps to needs to be changed from 66356 to 66357 to accommodate for this change. If we had other patches after this variable, they would similarly need to be shifted down in memory.

Originally we tried using one byte addresses, but found this would not be sufficient as we attempted to run larger test cases. In the end we opted for 3 bytes which was sufficient for the test cases and because it was a fixed size, patching was easier to implement. It was a priority for the project to translate as many features as possible for the compiler so memory efficiency was not a concern. This was also evident in the way we designed the layout of memory, such as using 1 word tags per value for composites in the heap. However, a downside of this simplification is that larger programs that require more than 3 byte jump addresses will not work. Future work could consider how this could be improved such as the creation of a generalised solution that varies patch size for jumps.

4.3 Language Features

This section is going to discuss the language features implemented in this project.

4.3.1 Primitive Operators

Primitive integer operations were fairly straightforward to implement. The following arithmetic operations have been implemented: *integer addition*, *integer subtraction*, *integer multiplication*, *integer division*, *integer remainder*. On the EVM, primitive arithmetic is executed using 256 bits, whilst primitive arithmetic is unbounded in Whiley. This project has assumed a fixed size for 256 bits for each Integer slot, which corresponds to the EVM wordsize of 256 bits in memory. There is no negation opcode in EVM, therefore we used subtraction from 0 instead.

More space is often taken up then needed for a particular value, such as with *boolean types* which take up one 256 bit word. However, only one bit would be necessary to represent these. This was a simplification made as memory usage was not a concern.

The following relational operators were implemented: *integer greater-than*, *integer less-than*, *integer greater-than-or-equal*, *integer less-than-or-equal*, *equal*, *not-equal*. As there are two conditions that need to be evaluated in *greater-than-or-equal*, and *less-than-or-equal*, at the bytecode level the two numbers that need to be compared are firstly duplicated before applying the two comparison opcodes followed by an `OR`. *Bitwise-and*, *bitwise-or*, *logical-not* were also implemented.

Short Circuit Evaluation

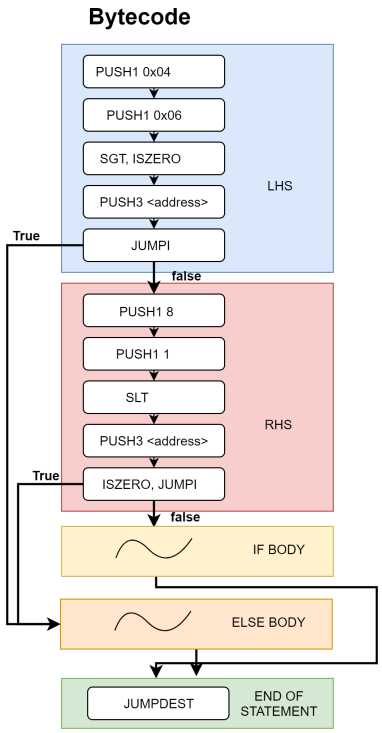

Figure 4.3: Short circuit evaluation CFG

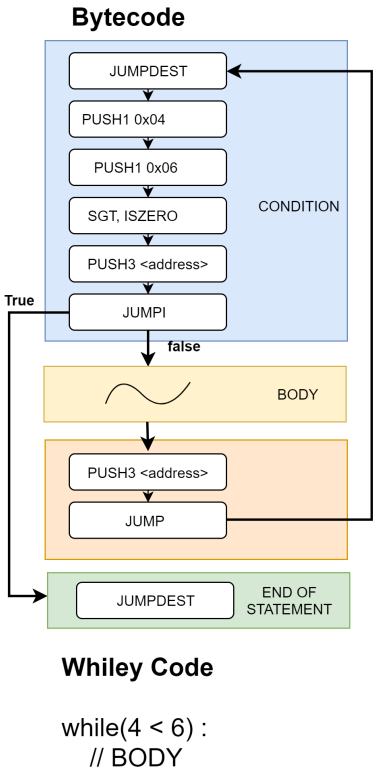

Figure 4.4: While Loops CFG

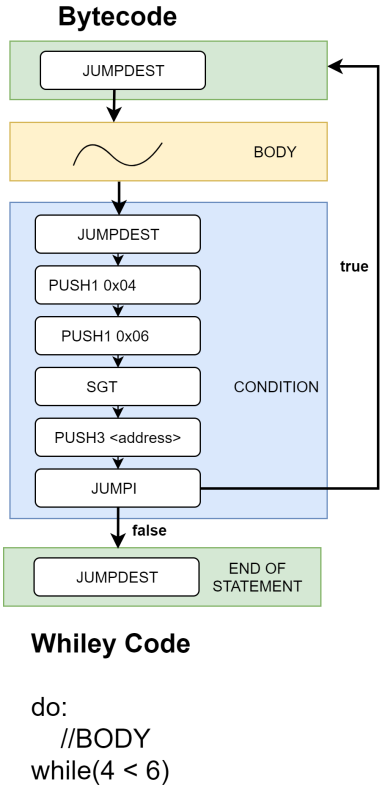

Figure 4.5: Do While Loops CFG

Logical-and and *logical-or*, were also implemented. These make use of short circuit semantics, which utilises branching based on whether the conditions are met or not. In the case of logical-and, this involves checking the value evaluated from the first expression, and branching out (not checking the other branch) if the first expression evaluates to false and returning false, otherwise checking the other expression and evaluating whether it also evaluates to true and returning true if so. This can be seen from the CFG in Figure 4.3, where the left hand side (LHS) of the expression checks whether 6 is greater than 4. If this isn't the case, the EVM will jump to the else body (note that `ISZERO` flips the bit, meaning a false evaluation of the condition results in a true jump), otherwise evaluates the RHS. In an *logical-or* condition, if the first expression evaluates to true, the EVM will skip the second expression and return true, otherwise will check the second expression and return what this expression evaluates to.

Short-circuit evaluation is needed for correct conditional evaluation as it means the second expression is only evaluated if necessary. This avoids the side effects that may come with the second expression, such as the invocation of a function for example, if it is not necessary to evaluate.

4.3.2 Control flow

If-statements, *while loops* and *do-while loops* were implemented in EVM. For *if-statements*, one patch is created that points to the target destination. In while loops, we keep track of two patches per loop, one to jump us out of the loop, and another one at the end of the loop body to jump to the beginning of the while loop condition to be evaluated. In the CFG in Figure 4.5, we first evaluate the condition `4 < 6` and if this is true, we enter into the body (as mentioned prior, `ISZERO` flips the bit so this is false for the jump), otherwise jump to the end of the statement. *Do-while loops* follow similar logic, however, make use of one patch. As the conditional is after the first execution of the body, the jump target for the patch is now on the start of the loop body, whilst the actual jump is evaluated at the end of the conditional below the loop body; which holds a jump target variable that is patched. We can see this from Figure 4.5 which first executes the body, before evaluating the condition to jump to the `JUMPDEST` above the body, or exiting out of the statement.

Breaks and Continues

Whiley has the concept of `break` or `continue` statements, which can be called in loop bodies to change the current state. The `break` statement results in the loop being exited, whilst the `continue` statement jumps to the beginning of the loop of the next iteration.

A challenge with patching `break` and `continue` statements is when jumping to the correct target you may have any number of nested blocks inside another nested block. Hence, there is a need to keep track of the nested blocks, so once a `break` or `continue` is reached, EVM can jump to the correct position at the start or end of the current block.

This was implemented within the compiler through abstracting the start and end of a new block that is visited to a concept called `Scope`. A stack composed of scopes are maintained. When entering a loop body, a new scope is created. When leaving a loop, this scope is popped from the stack.

Scopes record the start and end address of a loop body, as well as the different contexts that exist inside the body. These contexts refer to breaks, or continues, and can be extended to include others if required. Context objects store the context type, and the context memory address, which is where the context (`break` or `continue`) jump target variable exists on the stack. Figure 4.6 illustrates Whiley code that utilises two scopes, each of which represent a

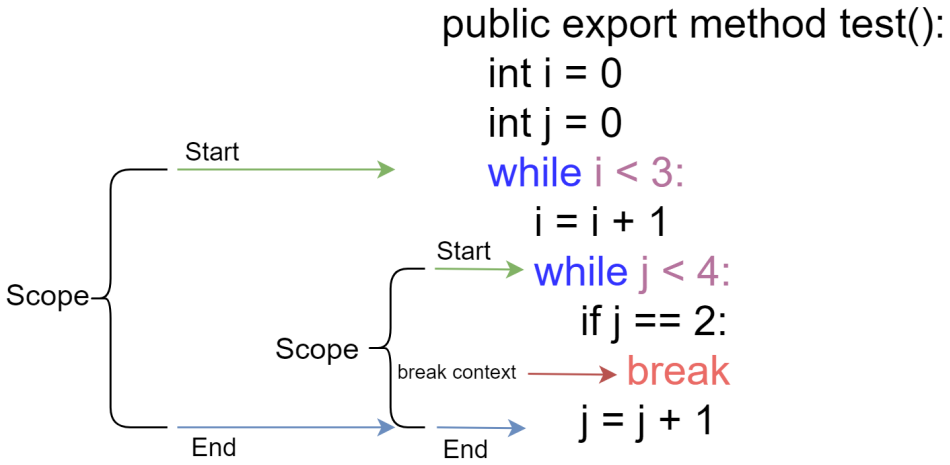

Figure 4.6: Scopes in terms of Code

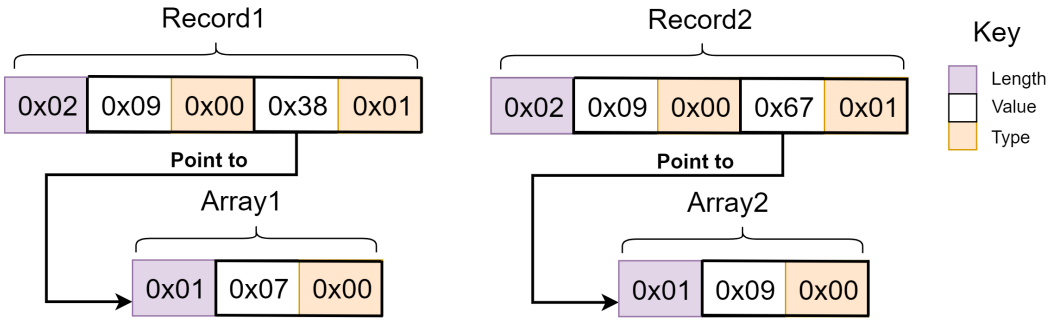

Figure 4.7: Examining two records for Equality

while loop, where one is nested inside the other. The position of the break is recorded in the scope, as well as the start and end of the scope block. These allow for the associated jump variable and target to be maintained.

Now that the jump positions for a block are known, which is either to the start or end of the block, a list of patches can be created. A continue jumps to the start position of the block in order to continue the next iteration of the loop. A break jumps to the end of the block to exit it. These patches are resolved the same way as with if, do-while and while, although in the case of these scope patches `JUMPDEST`s are not added as the break/continue statements jump to an existing jump destination. This would be above the start, or end of the loop body.

4.3.3 Equality

There is a need to check that two variables hold the same value. This is simple to achieve with primitive types. The two values can be pushed on the stack, and the `EQ` opcode can be executed which checks that these values are identical.

Checking that two composite data structures are equal is more complex however, as they are composed of multiple values manually maintained in the heap. This means that both composite structures have to be iterated over, and the values at each index compared. Furthermore, a composite structure may contain values that point to another composite structure so just evaluating every value at the surface level wouldn't be sufficient to check for equality. For example, consider Figure 4.7 which shows two records in memory. From the Figure we can see they are both the same length, and hold the same value 9 in the first

index. However, the nested arrays do not hold the same values, and hence would not be evaluated to equal. Although this may seem simple, to do so is a complex process in EVM bytecode, requiring multiple jumps to represent the different branches of the loops, as well as multiple temporary variables (held in scratch space) to keep track of the current iteration of the associated loop.

Equality and cloning are both implemented with inline functions in bytecode. This means that when composite equality or cloning is required, the bytecode for that operation will be translated into the current function, instead of storing the function in another section and dispatching to it. A benefit of this is that it is more efficient than having a single function to dispatch to which has the added overhead of the call and return which both cost gas in Ethereum. The tradeoff for this however is that there would be more bytecode created resulting in a larger program size as a program with multiple equals or clones would have an inline function for each of these.

The equals method was implemented by firstly comparing the size of both data structures. If their sizes are not the same, they are not identical, hence we jump out of the section of bytecode that handles the remaining checks and push 0 onto the stack to indicate equality between them evaluates to false.

The types at each index also need to be compared to ensure that they are in fact the same type. If the type at an index is indicated to be an array or record (storing a `0x01`), then the values in both composite structures referenced at that index need to be looped over and compared for equality, before jumping back to the main array or record to compare the rest of the values. In total, this has resulted in 11 jumps utilised for composite equality comparison, with variables that need to be incremented/decremented in scratch space.

Difficulties arose when implementing composite type equality in the AST as you do not necessarily know if the type of the variable you're comparing is a composite type or not. This is because you may be checking equality of a function and another function, so you have to check the return types of the functions to know whether to use the composite structure equality or primitive equality that makes use of a single `EQ`.

Because of the uniform representation of composites, both arrays and records use the same code for equality (and cloning). Furthermore, the not equals implementation reuses the equals bytecode and flips the bit at the end using the `NOT` operator.

A limitation of my implementation was that I only check the top level for nested composites. For instance, checking the values of an array or record inside a record, as we saw in Figure 4.7. However, if that record inside a record had another record inside of it, the values of that record would never be reached and checked. Hence, the implementation is limited to only jump to composites that are nested on the top level. In order to resolve this, we would have to allow such cases to generate an external function in order to support recursive operations for arbitrary level nested composites.

4.3.4 Cloning

Cloning was needed because compound data structures have value semantics in Whiley. This means that passing a compound data structure into a function will cause that structure to be cloned, similarly passing a primitive variable will result in a copy of that variable. For example, in the Whiley code in Listing 4.1, we see `rec` being passed into the function `f` which will result in a clone of it being created for use in `f`. The purpose of cloning is so that when we update a compound data structure inside a function, it doesn't affect the thing that called it.

Because Whiley uses pure functions, variables that passed in must always be copied. We chose to deep clone composites upon reaching a variable-copy node in the AST to ensure

```
{int a, int[] b} rec = {a: 9, b: [7]}
f(rec)
```

Listing 4.1: Passing the record `rec` into a function will result in a clone of `rec`

```
function checkBalances(int[] balances) -> int:
    return 1

public export method test() :
    int[] addr = [0, 1, 2, 3]
    assert checkBalances(addr) == 1
```

Listing 4.2: Variables are still cloned even when passed into functions that do not use them

that the safety of variables when modified in functions. Even if a variable is not read or written to, the Whiley compiler still indicates a variable copy is needed, so a clone of the composite is made. This can be seen in Listing 4.2 where the array `addr` is not read or modified in `checkBalances()`. However, the Whiley compiler will ask for `addr` to be copied, so a clone of this is made.

Eliminating copies at certain positions when allowed is an optimisation that is currently not made in the Whiley compiler but is being investigated [27, 30, 28]. Furthermore, there is no indication of when a shallow clone could be used instead of a deep clone so deep clones are always made.

Cloning in the EVM compiler is a similar though less complex process to equality in bytecode, that involves iterating over each value and making a copy at a new position in the heap. Similar to with composite equality, only addresses from the top level are examined, so at most a deep clone would only work for a two level structure. This however wasn't a concern as very few test cases utilized structures of more then 2 levels of nesting. Using a recursive function would help resolve this as discussed in the previous section.

Chapter 5

Evaluation

This chapter will examine the methods of evaluation and results of this project through the use of tests and examining memory usage.

5.1 Overview

The Whiley Compiler has an existing test suite made up of 642 Whiley files. Each of these files is a Whiley program that tests features of the Whiley language. These tests are not large programs but the test suite as a whole does test all the features of Whiley. An example of a test can be seen in Listing 5.1 which shows a Whiley program that tests while loops with array generators and array initialisers. Each Whiley test is translated into EVM bytecode before being executed on the EVM (we use the JEVM implementation as discussed). These Whiley tests check certain inputs and outputs using assert and assume statements, which when translated into EVM bytecode throw an exception on the EVM if the condition fails. If no such exception occurs and the bytecode evaluates without error, then an `OK` is returned from the EVM and we assume the test has passed. The test suite contains tests for all features of the Whiley language. These include features we're implementing such as arrays, records, whiles, do-whiles, but they also include many features we have not implemented such as bytes, coercion, function references, lambdas and more. This means that from the test suite, less than half the tests check for language features our compiler implemented. There are two kinds of test suites, the valid test suite which is made up of valid programs, and the invalid test suite which is made up of programs that do not verify. We aren't using the invalid test suite because we're only checking the correctness of our compiler.

Evaluation has been conducted using the Whiley test suite by creating JUnit tests for each of the existing Whiley test cases [33], and checking that they can be executed without error.

5.2 Methodology

This experiment was conducted by running JUnit tests in IntellJ on a Microsoft Windows 10 Pro x64 machine on a Surface Laptop with a Intel Core i5-7200U CPU @ 2.50GHz Processor and 8GB of RAM.

Programs in the test suite makes use of asserts and assumes, so these were translated into bytecode where the EVM will reach an `INVALID` opcode if the condition fails. The JUnit test will fail if a failed state such as this is reached. The test could also fail if an internal Java exception occurs when the JEVM attempts to execute the bytecode. These could include when attempting to access memory that doesn't exist (out of bounds), division by 0, attempting

```
function sum(int[] xs) -> (int r):
    int i = 0
    int sum = 0
    while i < |xs|
    where i >= 0 && sum >= 0:
        sum = sum + xs[i]
        i = i + 1
    return sum

public export method test():
    assume sum([1;0]) == 0
    assume sum([1,2,3]) == 6
```

Listing 5.1: `While_Valid_62`: Whiley test function for while loops with arrays

to execute an invalid bytecode, or if there is nothing on the EVM stack to subsume when executing opcodes such as `MSTORE` which requires two items on the stack. If a failed state or exception does not occur, the JUnit test will pass. This has allowed us to confirm that we can compile Whiley test cases to the Ethereum bytecode and that they will evaluate and execute correctly on the Ethereum Virtual Machine.

Memory usage was measured by recording out the number of words that each passing test uses in the EVM at the end of each program. The word count was added to get a total and divided by the number of passing tests to get an average word count. To get the memory usage in megabytes, the word count was multiplied by 256 bits (wordsize) to get a total memory usage in bits, before being converted into megabytes.

JProfiler is an industry grade Java profiler [4] that we used to gain further understanding of memory usage in our compiler and issues such as memory leakages. The profiler shows a variety of metrics such as Memory usage (top graph) and GC Activity (bottom graph) which can be seen in Figure 5.2.

We decided that measuring the time taken to run tests would not be particularly relevant to our project as we had not looked at optimization or efficiency. Furthermore, the JEVM library is quite a basic implementation of the EVM and is not specifically efficient. Hence, timing measurements would not provide any useful information that would help us evaluate this project.

To validate whether a test that had failed was expected to pass I manually went over each failing test, and created a table as shown in Figure 5.1. This table shows whether the test passes or not, and if we think this test should have passed. The reason we would expect a failing test to pass is because because it contained only features we had implemented. Other test cases that failed were expected to fail because they used features we hadn't implemented. Features we didn't implement include bytes, strings, addresses, global variables, open records, lambdas, unions, multiple returns, and more. Figure 5.1 shows us a section of the table where the first column shows the name of the test, the second shows whether the JUnit test passed or failed, and the third column shows whether I think the test should succeed or not. We can see on this graph the test `Array_Valid_3` which failed the unit test whilst I think it should have passed. For failed test cases I've also written down the features they use which may be making them fail to gain a better understanding of what is going on.

Test Name	Passing	Should Pass
Array_Valid_1.whiley	Fail	Fail
Array_Valid_2.whiley	Pass	Pass
Array_Valid_3.whiley	Fail	Pass
Array_Valid_4.whiley	Pass	Pass
Array_Valid_5.whiley	Pass	Pass
Array_Valid_6.whiley	Pass	Pass
Array_Valid_7.whiley	Pass	Pass
Array_Valid_8.whiley	Pass	Pass
Array_Valid_9.whiley	Pass	Pass
Assert_Valid_1.whiley	Fail	Fail
Assert_Valid_2.whiley	Pass	Pass
Assign_Valid_1.whiley	Fail	Fail
Assign_Valid_2.whiley	Fail	Fail
Assign_Valid_3.whiley	Fail	Fail
Assign_Valid_4.whiley	Fail	Fail
Assign_Valid_5.whiley	Fail	Fail
Assume_Valid_1.whiley	Fail	Fail
Assume_Valid_2.whiley	Pass	Pass
BoolAssign_Valid_1.whil	Pass	Pass
BoolAssign_Valid_2.whil	Pass	Pass
BoolAssign_Valid_3.whil	Pass	Pass
BoolAssign_Valid_4.whil	Pass	Pass
BoolAssign_Valid_5.whil	Pass	Pass
BoolAssign_Valid_6.whil	Pass	Pass
BoolFun_Valid_1.whiley	Pass	Pass
BoolIfElse_Valid_1.while	Pass	Pass
BoolIfElse_Valid_2.while	Pass	Pass

Figure 5.1: Table created that reports test passing status

5.3 Results

total passing	passing that should pass	array pass rate	record pass rate
39%	85%	69%	86%
251/642	251/293	85/124	44/51

When running the test suite, it was found that 251 tests JUnit tests passed out of 642, which is a passing rate of 39% for the Whiley test suite, which tests all language features of the language. Six of these test cases did not actually pass due to JEVM lacking the implementation for division and modulus. However, these were simple cases and appeared to generate the correct bytecodes so we are counting them as passing tests.

Based on the table in Figure 5.1 it was found that 42 tests that should be passing currently do not pass. This gives us a passing rate of 251/293 or 85% for the language features implemented. There are 124 tests that use arrays. 39 of those tests fail, giving us a failure rate of 31%. On the other hand, there are 51 tests that make use of records. 7 of those tests fail, giving us a failure rate of 14%. All failing tests that should pass utilise arrays, or records. All tests that only use other language features that we implemented pass.

We are unsure to the reason for a high array failure rate. However, a defining characteristic we noticed was issues in the reassignment of values in an array. For example, 8 of the 11 `ListAssign` tests which should be passing actually fail. These tests check assigning an array to that of another array. This suggests a bug exists within array assignment. An investigation into this suggests that although array assignment does result in a newly cloned array on the heap, attempting to modify the assigned array does not update the values. For example, consider Listing 5.3 which shows `arr2` being assigned to `arr1` which in Whiley results in a copy of `arr1` to be stored in `arr2`. Checking the values on the heap show this is successful. However upon checking the heap after the assignment the 3rd element of `arr2` to 2, we find that this array was not updated. Furthermore, hundreds of values storing `0x00` have now been appended to the heap, suggesting an update of some kind to an incorrect address. The reasons for this are unknown. Comparing this to Listing 5.2, the code `arr1[2] = 2` is successful and these issues are not present.

Although records and arrays seem to work well in simple cases with primitive types, such as in Listing 5.4, the implementation is fragile and including more structures and inter-

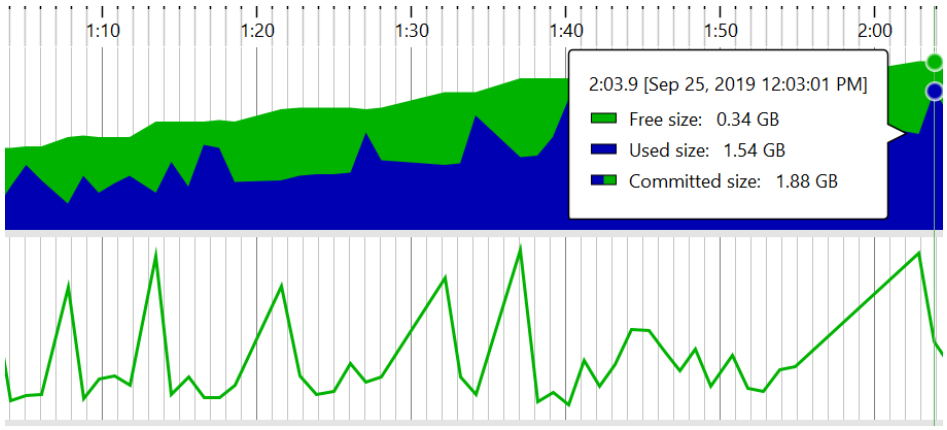

Figure 5.2: Memory Usage from Profiler

```
int[] arr1 = [4, 4, 4]
int[] arr2 = arr1
arr1[2] = 2
```

Listing 5.2: Array Assignment case 1: a passing test

```
int[] arr1 = [4, 4, 4]
int[] arr2 = arr1
arr2[2] = 2
```

Listing 5.3: Array Assignment case 2: a failing test

```
function f(int[] x) -> int:
    return |x|

public export method test() :
    int[][] arr = [[1, 2, 3]]
    assume f(arr[0]) == 3
```

Listing 5.4: A simple passing test of 2D arrays

```
type Point is { int x, int y }

function fromXY(int x, int y) -> (Point[] rs):
    return [Point{x:x, y:y}, Point{x:x, y:y}]

public export method test():
    Point[] ps = fromXY(1,2)
    //
    assert ps[0] == ps[1]
    //
    assert ps[0].x == 1 && ps[0].y == 2
```

Listing 5.5: A failing test

```
type mymethod is method()->(int)

public export method test():
    &int x = new 3
    mymethod m = &(->(*x))
    int y = m()
    assume y == 3
```

Listing 5.6: `Lambda_Valid_12`: False positve test

actions between them often result in test failure. For example, we would expect test case in Listing 5.5 to pass because records and arrays have been implemented. This test case fails for reasons unknown, the key characteristic we have observed is that this test contains a record inside an array. The test fails at both assert statements. I also noticed that the ordering of initialising composite data structures can also make tests fail so there may be some level of interaction between them due to incorrect references, such as with array assignment.

Listing 5.6 is an example of a false positive test that just happens to pass. Our compiler treats this code as assigning x to 3 and then assigning the method m to the value of x which is 3. Finally, y is assigned to m which sets it to 3. Lambdas have not been implemented in our compiler, however this test happens to pass unintentionally in this simple case. Examining the tests that do pass however, it appears that very few would actually be considered false positives.

5.3.1 Memory Usage

On average, 82 words in memory are used for each test as averaged over the 245 passing tests. As each word is 256 bits, that gives us an average memory usage of 2.6mb per test. This is quite a lot of memory for what are relatively small benchmark test cases.

There are a few reasons for this. Firstly, as composite data structures are not deallocated from the heap, having multiple compound data structures would utilize a lot of memory, especially considering that one word is allocated to represent the size, with two words for each value in the structure. Secondly, as functions are pure in Whiley, passing a composite structure into a function, would result in that structure being cloned on the heap. The same is applied to the return parameter of functions, although Whiley would reason about whether it should be cloned or not, this is almost always the case. Hence, it is not uncommon for tests that contain composite types to have 150 or more words in memory. This layout could be improved as we discussed in section 4.2.2.

However, some programs can be excessively sized. For instance, although the passing test `While_Valid_62` (Listing 5.1) tests arrays using while loops and function calls, 558 words are used. As this test only uses two composite structures and no cloning takes place, this test should be under 50 words. Examining the bytecode, there are hundreds of `0x00`s appended after the the final composite structure in the heap. It appears that somewhere an empty value/s are being pushed to a large address/s though examining the bytecodes we are currently unsure of where this is occurring. This bug has been spotted in other tests as well such as discussed with Listing 5.3. Although this does not directly affect a test case from passing, it results in an excessive amount of memory being utilised.

Furthermore, as the heap was set to begin at position 25, the fewest number of words that a program could utilise would be 26 (taking into account one heap pointer and unallocated scratch space). Simply allocating one composite type into memory would result in another 9 words being held for scratch space, as well as the composite type being allocated which

would leave us at 33 words being used at minimum (given an composite structure of one primitive value).

Extended Analysis

In order to further understand the issues related to memory usage highlighted in the previous section, we conducted further testing/investigation using a industrial strength Java profiler called JProfiler [4]. I ran JProfiler over all the test cases (both passing and failing) and we measured the memory output which is plotted in Figure 5.2. What we can see clearly from Figure 5.2 is that the amount of memory usage is increasing over time constantly. Furthermore, the Java Garbage Collection Activity (GC Activity) is abrupt. This suggests a memory leak of some kind.

5.4 Discussion

Testing provides us with some indication of how much of the language is covered from our project and which sections are not covered by it. This helps to highlight the focus of future work for the compiler, such as implementing new types such as union and nominal types, and fixing current issues with composite data structures.

Utilising a test suite allowed us to quickly validate the areas of the language covered by the project. However, as one specific language feature may be dominant for many of the tests, the current test percentage that is passing is not truly indicative of the proportion of the language covered. Nevertheless, testing still allows us to see areas of future development, what areas of the language may be infeasible for the current development cycle, and where our project succeeds. From the passing tests, we can see that many of the language features we have implemented do in fact pass although not completely accurate, whilst those that do not often use features that have not been implemented.

A limitation of our testing suite is that it provides little indication of how our compiled code would be with real Ethereum contracts deployed on a blockchain. The tests only tell us if the Ethereum code can execute on a simplistic implementation of the EVM. In the future, the compiler can also be evaluated through the creation of existing smart contracts, that can be run on the Ethereum blockchain.

The test suite also does not give us an indication of how optimised the bytecode is. Minimising gas usage when writing smart contracts is important. Hence, whilst producing correct EVM bytecode is important, if the gas usage is high due to un-optimized bytecode, it is unlikely that anyone would want to use our solution.

As every Whiley file that is provided to the test suite is valid, a successful test indicates that the compiler succeeded in translation, and has not failed during execution. However, it cannot indicate whether the bytecode produced is actually correct. However, at present false positives do not present an issue; rather, the test suite as a whole is used to get an idea of what works and what doesn't.

5.5 Case Study

This section will prevent a case study for writing Ethereum smart contracts using Whiley. We will presented two approaches to this.

One of the biggest challenges in representing Ethereum smart contracts in Whiley is representing the contracts storage. This is the state that persists across invocations of a contract. Reading and writing to contract memory and storage in Ethereum is expensive. Therefore,

```
1    function deposit(Contract c, Msg msg) -> (Contract, int):
2        ...
3        return c, c.balances[msg.sender.value]
```

Listing 5.7: Functional purity requires you to return the whole array

using pure functions would be incredibly expensive as they do not modify the data you pass in as parameters, instead data is allocated for a new variable. To put this into context, if we had a contract with an array of one thousand accounts and a function `deposit()` that was passed this array as a parameter, updating an element of the array using this function would result in a new array of one thousand accounts with only one element being updated. As functional purity does not let you know which elements you have updated, all one thousand elements would need to be written into contract storage. This would be extremely expensive and is not realistic. This can be seen in Listing 5.7 where the parameter `Contract` that holds an array of balances, and `Msg` both need to be cloned upon being passed into the function `deposit()`, and when these variables are returned require all values within them to be written into Ethereum storage.

This is the motivation for using Whiley methods in our case study. Method are comparable to methods in Java and allow impure features, in particular, updating state through references. We need methods so we can update the contract storage in place, so when we have a write through a reference it will correspond to a single write under Ethereum bytecode. We recreated the simple bank contract from Listing 2.3 as would be imagined in Whiley. This contract would not compile with my compiler in its current state as it uses addresses (&), and global variables. Furthermore, the JEVM library has not implemented contract storage yet (just stack and memory), so realistic contracts can not be created yet.

There are two approaches to creating this smart contract in Whiley. These are labelled Approach A, and Approach B, which correspond to Listing 5.9 and 5.10 respectively. Listing 5.8 shows the declaration of record types that both approaches use.

Approach A

Approach A uses globally stored variables in a similar fashion to Solidity. These global variables will need to be connected so that when updating the global variables, Ethereum bytecode would be generated to store it into the contracts storage. This will require subsequent updates to Whiley as currently global variables in Whiley are final so cannot be updated after initialisation. This is because in Whiley you want to be able to characterise in specification what side effects you can have (what things can change). However, there is no support for this in Whiley at this time. However if it did it would look like what we see in Listing 5.9, which shows the simple bank contract using global variables. Notice that each method in Listing 5.9 and Listing 5.10 contain pre and post conditions (requires and ensures), which helps Whiley verify about code in those methods and is the motivation for writing smart contracts in Whiley. The methods in these Listings are preceded with the keywords `public export`. `public` means other Whiley files can access this method. `public export` means things outside of Whiley can access this method, such as the Ethereum system. The idea of preceding methods with this identifier is to allow others to interact with Whiley files externally such as Solidity contracts.

```
type Msg is { int value, Address sender}
type Address is { int value }
type Contract is {int[] balances, int balance}
type Msg is { int value, Address sender}
```

Listing 5.8: Header Information

```
Address owner
int clientCount
Msg msg
Contract storage

public export method constructor()
requires msg.value == 30
ensures clientCount == 0:
    clientCount = 0

public export method enroll() -> (int z)
requires msg.sender.value >= 0
requires msg.sender.value < |storage.balances|
ensures clientCount <= 3 && z >= 0
ensures z == storage.balances[msg.sender.value]:
    if clientCount < 3:
        clientCount = clientCount + 1
        storage.balances[msg.sender.value] = 10
    return storage.balances[msg.sender.value]

public export method deposit() -> (int z)
requires msg.value >=0 msg.sender.value >= 0
requires msg.sender.value < |storage.balances|
ensures z == balances[msg.sender.value] + msg.value:
    balances[msg.sender.value] += msg.value;
    return  balances[msg.sender.value]
```

Listing 5.9: Recreation of Simple Bank Smart Contract with global fields: A

Approach B

Approach B is an alternative way to create smart contracts in Whiley that is currently supported within the language. The idea is to represent contract storage using references so updating references updates the contract storage. These references can be seen in the method parameters in Listing 5.10, and are identified by `&`. These references point to an area in contract storage. Writing into a reference would be set to write into contract storage (using Ethereum bytecode, e.g. `SSTORE`). A disadvantage of this approach is it requires you to pass each parameter into a method when you want to utilise variables in storage, potentially leading in large method headers.

```
1     public export method constructor(&clientCount, &Msg msg)
2     requires msg.value == 30
3     ensures clientCount == 0:
4         clientCount = 0
5
6     public export method enroll(&Contract storage, &Msg msg, &
          int clientCount) -> (int z)
7     requires msg.sender.value >= 0
8     requires msg.sender.value < |storage.balances|
9     ensures clientCount <= 3 && z >= 0
10    ensures z == storage.balances[msg.sender.value]:
11        if clientCount < 3:
12            clientCount = clientCount + 1
13            storage.balances[msg.sender.value] = 10
14        return storage.balances[msg.sender.value]
15
16    public export method deposit(&Contract storage, &Msg msg)
          -> (int z)
17    requires msg.value >=0 msg.sender.value >= 0
18    requires msg.sender.value < |storage.balances|
19    ensures z == balances[msg.sender.value] + msg.value:
20        balances[msg.sender.value] += msg.value;
21        return  balances[msg.sender.value]
```

Listing 5.10: Recreation of Simple Bank Smart Contract: B

Discussion

From both approaches we can see the use of requires and ensures clauses which enforce conditions the Whiley compiler can verify. The preconditions of a method are important as it allows Whiley to verify the code within the method. However, there is no way of knowing if a precondition is met if externally called. As discussed, public export methods can be externally called from non Whiley files. Therefore, as Whiley cannot determine the state of the program at this point, these preconditions will need to be translated into EVM bytecode so they can be checked at runtime. Internal method specifications do not need to be written into bytecode as you know the caller within the Whiley file, and is how the compiler currently verifies.

Post conditions could also be encoded into EVM bytecode. However, they are not as important as the preconditions for external calls as if you can require a certain precondition to be met, the Whiley compiler has enough information to reason about the exit state of the method. However, post conditions have the potential to be useful for run-time testing, so new syntax could be added that allows you to specify whether you want your post condition to be encoded into EVM bytecode or not. Ideally you wouldn't want to translate conditions that aren't useful such as post conditions into EVM bytecode, as every bytecode executed on the EVM costs gas. Other internal specification elements such as loop invariants could also be optionally translated for the purposes of debugging/testing at runtime.

Chapter 6

Conclusions

The goal of the project was to compile Whiley to Ethereum bytecode to utilise Whiley's verification capabilities to verify smart contracts. We achieved this by translating some of the major Whiley language features such as primitives, control flow, arrays and records into Ethereum Bytecode. We began this report by presenting some background on the current state of Ethereum smart contracts, common vulnerabilities on the EVM, and existing solutions. We then discussed the design of the compiler, in particular, dispatching, and the layout of memory. Our implementation was then discussed. Some challenges of the implementation was the layout of data structures in memory and equality and cloning of composites on the heap. Finally, we evaluated the compiler using a test suite, and found that although some problems still exist within the compiler, a reasonable number pass of tests pass. Although arrays and records still need bug fixing, this project has been successful in setting the foundation for a Whiley to Ethereum bytecode compiler.

6.1 Future Work

Firstly, existing problems with the implementation of Arrays and Records need to be fixed. This is both to make the state of the compiler more stable, and to allow the compiler to work with more complex use cases. As composite types are incredibly fragile, if future development was done, it may be a good idea to begin by re-implementing them instead of building on this solution. Furthermore, it may be best to re-implement the tag system for composite types to use one word for the tags, as suggested in section 3.3, to reduce memory usage.

One of the goals of creating a Whiley to EVM compiler was to use Whiley's verification capabilities to make writing smart contracts safer. Whilst this was investigated at a theoretical level, there is a need to write realistic EVM smart contracts with Whiley to see how well Whiley performs. This will require extensions to the compiler to support references, and global variables, as discussed in the section 5.5.

Currently the Whiley verifier does not correctly reason about references in methods as it cannot tell whether you've updated a reference, so improvements will need to be made. Furthermore, preconditions for public export methods will need to be translated into EVM bytecode so these methods can be verified at runtime.

Work also needs to be completed to translate more aspects of the Whiley language into EVM, so more complex use cases can be modelled. Whiley supports a number of different types, so the next choices would be union types, overloading, addresses, open records, global variables, before moving onto more complex types such as lambdas.

Ethereum specific language implementations will be needed in Whiley to make it useful

for writing Ethereum smart contracts. Currently there are many EVM bytecodes that cannot be connected to any Whiley feature, such as `BLOCKHASH`, `GASLIMIT`, `GASPRICE` and many more. This may require the creation of a low level API (library) which allows for native EVM functionalities similar to Solidity. This API could declare these as native methods which are methods without a body in Whiley. By not having a body it is expected that the platform would provide that body, in order words, during the translation into EVM bytecode.

Because there is no way of invoking a function of an external contract, the current ABI will need to be improved to implement the Solidity ABI. This will allow you to interact with Solidity contracts as well as find the information needed to dispatch to any function within a Whiley contract. Therefore, there is a need for an ABI that works between contracts (not just within a contract).

6.1.1 A comparison of existing solutions

The paper 'Precise Attack Synthesis for Smart Contracts' [15] shows us a comparison table of the vulnerability support of the different tools for EVM smart contract analysis, as seen in Figure 6.1.

Tool	Generate Exploit?	Common Vulnerabilities								
		Reentrance	Arithmetic	DoS	Bad Random	Timestamp	TOD	Unchecked Calls	Access Control	Short Address
OYENTE [11]	◐	✓	✓			✓	◐			
Mythril [32]	◐	✓	✓			✓	◐	✓	✓	
Zeus [15]		✓	✓			✓	◐	✓	✓	
teEther [16]	✓									
Securify [12]		✓				✓	◐	✓	✓	
MADMAX [14]			◐	◐						
ContractFuzzer [13]	✓	✓			✓	✓	◐			
SMARTSCOPY	✓	✓	✓	◐	✓	✓	◐	✓	✓	✓

TABLE II: A Comparison of Existing Tools for Smart Contract (Order by publish date). ◐ represents limited support.

Figure 6.1: A comparison of tools in the detection of common vulnerabilities [15]

Below is how we have imagined Whiley could be used to spot existing contract bugs.

Reentrancy Bug. The reentrancy bug could be avoided through Whiley's verification capabilities. The specification of a function that transfers ether could enforce that that one account would have a specified increased amount, with the other account would having their account decreased by the same amount. This would mean that a called function that attempts to recall the current function would not be permitted due to the exit condition not being met. For this to work in Whiley, the verifier will need to be improved so that it can figure out if an external contract call will make the specification fail, as currently Whiley will always assume it will fail.

Arithmetic bugs. Whiley Specifications would limit the range of values to help prevent arithmetic bugs such as integer overflows or underflows. It is likely Whiley will be able to detect these bugs.

Denial of Service (DoS). It is unlikely that Whiley could be used to detect Denial of Service vulnerabilities in a contract as the logic of a contract would need to be modified to avoid this, rather than verification conditions. However, this could be an area of future investigation.

Timestamp Dependance. Although EVM specific features do not currently exist in Whiley, once a library is produced that allows the user to use the timestamp, the Whiley verifier could suggest warnings for when the timestamp is used to generate values (e.g. random numbers) used to modify state. However, it would not prevent misusing the timestamp as it is in the programmers hands to make this does not happen.

Transaction Ordering Dependance (TOD). It is possible that Whiley would be able to avoid timestamp dependence attacks. This would require that specifications were encoded into EVM bytecode, as currently specifications are only validated at compile time. Using these specifications, if the value of something is not what is expected, the transaction can be set to fail, whether or not an attacker changed the state of the program prior to the block being mined. Hence, the contract will fail upon specification not being met, as there is an expectation of a certain state.

Unchecked Calls. Using preconditions and postconditions (requires and ensures) with methods in Whiley, calls can be ensured to return some desired state. Encoding these conditions in EVM bytecode would help to ensure that this works at runtime.

Other vulnerabilities. How useful Whiley would be at detecting Other Ethereum vulnerabilities is a topic of future investigation. It may be the case that modifications would be made to Whiley to support detection of other vulnerabilities, or the translation can be designed to support better practices. For example, Whiley could have a safe fallback function by default, which could prevent the misuse of the fallback function. Unlike with MadMax, Isabelle/HOL, ZEUS, and others, our project does not face the difficulties of CFG flow-analysis, as Whiley provides static analysis on the Whiley language level. Whether this could be in fact limiting could be a topic for future research.

Bibliography

[1] Units and globally available variables, 2018. https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html.

[2] Bitcoin script, 2019. https://en.bitcoin.it/wiki/Script.

[3] Contract abi specification, 2019. https://solidity.readthedocs.io/en/v0.5.12/abi-spec.html.

[4] Jprofiler: The award-winning all-in-one java profiler, 2019. https://www.ej-technologies.com/products/jprofiler/overview.html.

[5] Known attacks, 2019. https://consensys.github.io/smart-contract-best-practices/known_attacks.

[6] AGGARWAL, S. Understanding ether vs gas, 2017. https://conspirat.us/understanding-ether-vs-gas-82ce2f1dc560.

[7] ALTHAUSER, J. Bitcoin's Price Surpasses $18,000 Level, Market Cap Now Higher Than Visa's. https://cointelegraph.com/news/bitcoins-price-surpasses-18000-level-market-cap-now-higher-than-visas.

[8] AMANI, S., BÉGEL, M., BORTIN, M., AND STAPLES, M. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2018), ACM, pp. 66–77.

[9] ATZEI, N., BARTOLETTI, M., AND CIMOLI, T. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive 2016* (2016), 1007. http://eprint.iacr.org/2016/1007.

[10] B, M. Arithmetic overflow/underflow for smart contract security, 2019. https://www.nvestlabs.com/2019/04/23/arithmetic-overflow-underflow-for-smart-contract-security/.

[11] BANISADR, E. How $800k evaporated from the powh coin ponzi scheme overnight, 2018. https://blog.goodaudience.com/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530.

[12] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., GOLLAMUDI, A., GONTHIER, G., KOBEISSI, N., KULATOVA, N., RASTOGI, A., SIBUT-PINOTE, T., SWAMY, N., ET AL. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (2016), ACM, pp. 91–96.

[13] CHRISETH. Version 0.4.16, 2017. https://github.com/ethereum/solidity/releases/tag/v0.4.16.

[14] FALKON, S. The story of the dao - its history and consequences, 2017. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee.

[15] FENG, Y., TORLAK, E., AND BODIK, R. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067* (2019).

[16] GARCIA, R. Exercise: Simple solidity smart contract for ethereum blockchain, 2018. https://www.codementor.io/rogargon/exercise-simple-solidity-smart-contract-for-ethereum-blockchain-m736khtby.

[17] GRECH, N., KONG, M., JURISEVIC, A., BRENT, L., SCHOLZ, B., AND SMARAGDAKIS, Y. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018), 116.

[18] HOLLANDER, L. The ethereum virtual machine - how does it work?, 2019. https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e.

[19] KALRA, S. Ndss 2018 zeus: Analyzing safety of smart contracts, 2018. https://www.youtube.com/watch?v=X1_CoaIQ0SU.

[20] KALRA, S., GOEL, S., DHAWAN, M., AND SHARMA, S. Zeus: Analyzing safety of smart contracts. In *NDSS* (2018).

[21] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), ACM, pp. 254–269.

[22] MILLER, A. contract can lose the funds!, 2015. https://github.com/etherpot/contract/issues/1.

[23] MILLER, A. Report: Security audit of btc relay implementation, 2015. http://soc1024.ece.illinois.edu/BTCRelayAudit.pdf.

[24] NAKAMOTO, S., ET AL. Bitcoin: A peer-to-peer electronic cash system.

[25] NASHATYREV, R., ET AL. ethereumj, 2019. https://github.com/ethereum/ethereumj.

[26] PALANISN, ET AL. Anonymous function in solidity example code ask, 08 2017. https://ethereum.stackexchange.com/questions/24439/anonymous-function-in-solidity-example-code/24443.

[27] PEARCE, D. Improve moveanalysis phase, 2017. https://github.com/Whiley/WhileyCompiler/issues/798.

[28] PEARCE, D. Unnecessary cloning operations, 2017. https://github.com/Whiley/Whiley2JavaScript/issues/23.

[29] PEARCE, D. Jevm, 2018. https://github.com/DavePearce/JEVM.

[30] PEARCE, D. Clone optimisation move analysis, 2019. https://github.com/Whiley/Whiley2JavaScript/issues/34.

[31] PEARCE, D. Getting started with whiley, 2019. http://whiley.org/download/GettingStartedWithWhiley.pdf.

[32] PEARCE, D. Whiley (programming language), 2019. https://en.wikipedia.org/wiki/Whiley_(programming_language).

[33] PEARCE, D. Whileycompiler, 2019. https://github.com/Whiley/WhileyCompiler/tree/develop/tests/

[34] PEARCE, D. J., AND GROVES, L. Designing a verifying compiler: Lessons learned from developing Whiley. *Science of Computer Programming 113 (part 2)* (Dec. 2015), 191–220.

[35] SHAW, R., ET AL. Call stack, 2019. https://en.wikipedia.org/wiki/Call_stack.

[36] UNKNOWN. Post-mortem investigation (feb 2016), 2016. http://www.kingoftheether.com/postmortem.html.

[37] UNKNOWN. Contracts, 2019. https://solidity.readthedocs.io/en/v0.5.3/contracts.html.

[38] WEN, Z. A., AND MILLER, A. Scanning live ethereum contracts for the "unchecked-send" bug, 2016. http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[39] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper 151* (2014), 1–32. https://ethereum.github.io/yellowpaper/paper.pdf.

[40] YU, A. Ethereum development tutorial. https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial.