ESBMC BASED ETHEREUM SMART CONTRACT SECURITY VERIFICATION

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Student id: 10206402

Department of Computer Science

Contents

Word Count: 13552

List of Tables

List of Figures

Listings

Abstract

ESBMC BASED ETHEREUM SMART CONTRACT SECURITY VERIFICATION

Burhanuddin Salim

A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2022

The aim of the thesis is to investigate and extend the smart contract vulnerability detection capabilities of ESBMC Solidity, an SMT based bound model checker. The project extends the current supported grammar to structs, constructors, built in globals, and extend the integer types supported. Moerver, the project also aims to and delivers to implement the ability for ESBMC Solididity to detect well known smart contract vulnerabilities, namely, Reentrancy, authorization through tx.origin, and integer overflows and underflows. The implementations are compared against well establish industry tools, namely, Mythril, Slither, Smartcheck, and Remix IDE. A benchmark of tests is designed and evaluated with to depict comparisions between the work of this thesis and industry standard tools. It is observed that the implementations allow for ESBMC Solidity to prove more performant and better cover detection of vulnerabilities that this is thesis is concerned with.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Dr. Cordeiro for his support and guidance both academically and in terms of project management. Also, I would like to extends aspecial thank you the ESBMC team who were alwasys available to support my development journey with ESBMC.

Chapter 1

Introduction

1.1 Motivation

Blockchain technology has amassed a great following in today's world both among financial and technological enthusiasts. It serves as the primary technology upon which cryptocurrencies like Bitcoin are based upon. It has also opened up an avenue for the likes of Ethereum to be able to leverage the decentralised nature of the blockchain technology and create protocols to enable decentralised computing by the use of smart contracts.

Solidity is a popular choice[32] among Ethereum[43] smart contract developers as a programming language to write smart contracts in which has domain specific features like sending ether to addresses which make the development of the contract much easier. This contract is then compiled to an EVM[43] (Ethereum Virtual Machine) bytecode representation, which then then evaluated by the validators on the network when the contract is published to the Ethereum mainnet.

The immutability of these contracts is absolute, they cannot be amended by any entity be it the user, developer of the contract or some admin entity for the network which makes the presence of vulnerabilities much more difficult to mitigate. The immutability of the contract begs the need for tools that can analyse smart contracts and detect vulnerabilities in the contract before it is published to the blockchain.

A well known hack within the blockchain community is one called the DAO attack[31], which incurred a loss of about $60 million forcing the ethereum network to rollback the blocks to a point before the attack. As of the writing of the paper organizations are paying hefty sums to those who are able to find vulnerabilities in their smart contracts, this includes the likes of Ethereum paying upto $250,000[1], 0xProject paying upto $1,000,000[2] for critical contract bugs.

1.2 Aim & Objectives

1.2.1 Aims

This project aims to further the attempt of ESBMC-Solidity to verify smart contracts by expanding the grammar support and also adding utilities to detect common vulnerabilities found in smart contracts. It aims to do so in a performant manner such that it measures well against current industry tools.

1.2.2 Objectives

1. Assess the ESBMC codebase to understand the current scope of the tool
2. Investigate the grammar and syntax of solidity and enable ESBMC-Solidity to verify through contracts that possess complex grammar features, namely structs, more primitive types, constructors, and solidity built in globals.
3. Investigate common smart contract vulnerabilities and design a method using ESBMC to detect those vulnerabilities, namely, Reentrancy, authorization through tx.origin, and Integer overflows and underflows.

[1]`https://ethereum.org/en/bug-bounty`
[2]`https://docs.0x.org/developer-resources/bounties#bounties`

Chapter 2

Background

2.1 Blockchain & Smart Contracts

2.1.1 Blockchain and Bitcoin

The blockchain technology today is described as a decentralised distributed ledger[28]. The means to achieve this technology are grounded in ideas that date back to 1991, when [25] outline a time stamping service which aims to timestamp digital documents. The service described, would allow the preservation of the order of digital documents by issuing a certificate signed by the service which includes a timestamp and a pointer to the digital document that precedes the document that is to be certified. The nature of the pointer is such that it is sensitive to the data that it points to, if the data is invalidated then so is the pointer itself which can be achieved by leveraging cryptographic hash functions[41] on the data stored at the pointer[25]. This allows any verifier to ensure that the order of the documents is correct.

Bitcoin[36] used this timestamping of documents along with a consensus protocol to create a digital currency where the wealth of a user is not recorded with any physical token but by a decentralised ledger. This idea of a set of documents being linked and by a certificate to the previous documents issued forms the basis of the idea of a (virtually) tamper-proof ledger that is used by Bitcoin . The transactions on the Bitcoin blockchain imitate the documents above but instead of being linked one transaction after the next transactions are grouped together in blocks in order which are then linked to each other. Each block possesses the hash of the data of the block before, the timestamp of the block, the ordered transaction data, and other meta data like version and difficulty index. Each block also has a nonce that is a part of the computationally

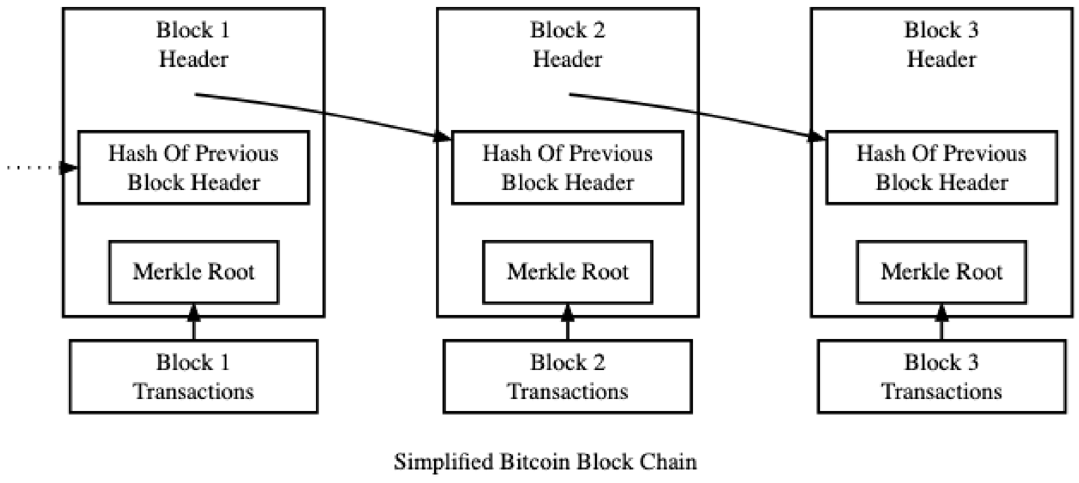


Figure 2.1: Graphical depiction of the Bitcoin blockchain taken from the bitcoin developers documentation[1]

intensive task that is a part of the Proof of Work system. (Figure 2.1).

The timestamping allowed it to maintain a correct order of transactions but to make it decentralised a consensus protocol[36] has to be designed for a peer-to-peer network. This protocol would dictate valid transactions and valid blocks that can be added by anyone onto the chain along with the proof of the solution of a computationally intensive puzzle (Proof of Work). Those that would make the computational effort to do so are known are miners as by doing so they collect a certain amount of bitcoin for themselves for a successfully constructed block.

By having each block to be backed by significant computational effort this lowered the chances of malicious miners putting in illegitimate transactions as they would have to have a significantly larger amount of computational power[44] to keep adding blocks after the malicious block so it will be accepted by the network as they are racing against other miners to validate transactions.

2.1.2 Ethereum and Smart contracts

This idea of the blockchain was then taken further to achieve a programmable blockchain by Ethereum[43]. Ethereum leveraged the blockchain technology that Bitcoin utilised to develop a programmable blockchain where the blockchain transactions would not only be transactions of currency between users but could also be function calls by users to user authored programs that would execute procedures from the program manipulating the state of a program. The currency used on the Ethereum blockchain is known as ether[6].

These programs are stored on the blockchain too and are dubbed smart contracts by Ethereum. These contracts can be called by users but also by other smart contracts by simply referring to their address on the blockchain. As the smart contract lives on the blockchain, it inherits the absolute immutability of the blockchain which is that once a smart contract is published it can no longer be amended or deleted by any party as this would invalidate the chain of blocks that succeed the block with the smart contract. A contract is able to perform a multitude of actions, including but not limited to: sending and receiving ether, calling functions on other contracts, storing and updating state. Each operation in a smart contract is deterministic and this is necessary as the state of the contract is not stored but rather can be retrieved from the transaction history of function calls made to the contract which need to make the same state changes every time otherwise the state can be different for different viewers of the chain.

As the computing is decentralised there needs to be a consistent execution environment that is unaffected by the architecture of the node that it is running on this is to ensure the determinism of the code. In efforts to achieve that Ethereum has developed a virtual machine named Ethereum Virtual Machine[43] which forms a framework that can be followed when executing smart contract code. All Ethereum smart contracts are compiled down to EVM bytecode and stored on the chain as EVM bytecode. This, however, is too low level for development and smart contracts are written in higher level languages the most popular of which is Solidity[32].

An example use of a smart contract is that of a crowd funding contract shown in Listing 2.1. The contract body, which is accessible publicly, outlines the terms of the crowd funding detailing how funds are pledged how they are collected at the end of the fundraiser. It contains 3 functions `pledge`, `finish` and the `constructor`. To begin with the beneficiary of the fundraiser publishes the contract which sets the beneficiary as the owner of the contract and sets the value they give as the goal and end for the fundraiser by the execution of the `constructor`.

The values for goal, raised and end are publicly available and can be queried by any user indicated by public on the variable definitions. The process for a `pledge` is also dictated it checks if the pledge is made before the end and then both the total raised value and the supporter pledge amount is updated in a hashmap of sorts. Once the fundraiser is over then the owner can call the `finish` method and if the goal has been reached then they can take the funds (selfdestruct sends all the funds of the contract to the argument address). Also note, there is no refund method which means that there is no way to recall your funds if you have a change of heart once pledged, but this is transparently outline by the absence of any such procedure.

The beauty of this is, firstly, the entire terms and procedure of the sale are transparent and no implementation details are hidden from the user. Secondly, the exchange requires no trusted third party, when a transaction for any of the functions is called both parties, the beneficiary and the benefactor, will be credited or debited according to the procedure transparently outlined in the contract and the trust on the transaction is provided by the consensus protocol without any trust between the parties themselves. Lastly, the terms of the exchange will never change, regardless of when the `finish` function is run it will behave exactly as it is outlined at the publishing of the contract.

To further elaborate, each transaction on the the Ethereum block chain has a few builtin values when it is constructed like the value of ether that is being sent with the call, the amount of gas being sent with the call, the address of the caller, parameters of the function call, contract address where the function lives and additional meta data. For every transaction the gas sent is an amount of ether sent for the miner who verifies the transaction, it is intended to incentivise miners and compensate them for the computational effort that goes to run the function call.

2.2 Smart contract vulnerabilities

As mentioned before Ethereum smart contracts are immutable objects and cannot be amended once publish. This heightens the severity of bugs that would otherwise be less severe due to easy remediation. Moreover, smart contracts are often responsible for large amounts of ether[16] which further elevates likelihood of malicious actors targeting smart contracts and raising the severity of bugs even more. Failure to successfully audit the security of smart contracts has often caused large financial losses in the millions, like that of the infamous DAO attack[31] which leveraged a reentrancy attack to siphon off funds in the neighbourhood of \$60 million, Parity's Multi-signature Wallet Hack[37] which also succumbed to a reentrancy attack and suffered losses of about \$30 million.

In recent years many reviews[8, 37, 38] have been conducted to assess the state of security of Ethereum smart contracts which highlight common vulnerabilities and attacks that are employed against smart contracts. Security vulnerabilities are so common that a study[27] during 2016 estimates that of all the published contracts about 45% of them are affected by known security vulnerabilities. This project focuses on

```
pragma solidity 0.8.13;
contract CrowdFund {
  mapping(address => uint) supporters;

  uint public goal;
  uint public raised;
  uint public end;

  address payable owner;

  constructor(uint _goal, uint _end) {
    owner = msg.sender;
    goal = _goal;
    end = _end;
  }

  function pledge() payable public {
    require(now < end);
    supporters[msg.sender] += msg.value;
    raised += msg.value;
  }

  function finish() public {
    require(now > end);
    require(goal <= raised);
    require(msg.sender == owner);
    selfdestruct(owner);
  }
}
```

Listing 2.1: An example crowd funding contract written in Solidity

bugs that arise when programming smart contracts in Solidity.

The developer community has also put in efforts to collate a database[1] of vulnerable contracts and known vulnerabilities to further help enhance the security of the smart contract ecosystem. One of such efforts is know as the SWC Registry[3] which stands for Smart Contract Weakness Classification and Test Cases Registry, it collects a set of of known vulnerabilities, remediations and example vulnerable contracts to prove as a reference for block chain developers. Below, some of the most popular and most discussed vulnerabilities are defined and explained.

2.2.1 Integer arithmetic

With everyday mathematical integer arithmetic one is safe to assume that given two positive integers the sum of those integers will be positive and an even stronger implication is the sum will be greater than either of the two integers. This logic can be implemented given that integers are stored with an infinite bit width, however, as modern computing has a finite memory it chooses to represent integers with finite bit width types. Along with the finite bit width representation for integers comes many caveats, the logic that the addition of two positive integers is positive no longer is guaranteed and is implementation specific based on the machine, the compiler and various other execution details. Let x be represented by a 4 bit integer, now suppose, $x = 15$ then in a general case it is ambiguous what is to happen if you add 1 to x either the operation can be leave the value unchanged or the addition can cause the value to go to a fixed value. Different languages define or choose to mark as undefined when addition is performed beyond the maximum value of the type. For C the standard[26] defines that for unsigned integer types a wrap around is performed where addition would be performed modulo the maximum value + 1, but for signed integer it is notes that it causes undefined behaviour.

Integer arithmetic beyond the maximum value is known as integer overflow. Analogously, when integer arithmetic proceeds in the other direction an integer underflow can occur and the values can wrap around the other way. In Solidity, a lot of different signed and unsigned integer types are made available each have a fixed with ranging from 8 to 256 in increments of 8[5]. As the execution of any contract is deterministic the cases for overflow and underflow are both defined to be wrap around where the value simply wraps over to the smallest value when overflowed and the largest value

[1]`https://github.com/crytic/not-so-smart-contracts`

when underflowed.

A definition of the state change with underflow or overflow is very helpful in thwarting bugs as the state change is well defined and the developer can predict the execution of the program. However, the bug arises when a developer fails to recognize and causes logic to be dependant on the fact that addition or arithmetic in general works the same as infinite bit width arithmetic. Listing 2.2 depicts a contract vulnerable to an attack due to integer overflow.

Listing 2.2 is a TokenSale contract with allows the buying and selling of tokens with a fixed exchange rate, it has been adapted from the Token Sale challenge[30]. On first glance, the contract allows a user to buy and sell the contract defined token at a constant exchange rate of 1 ether. To buy a token the caller must send across the value of ether corresponding to the number of tokens they would like to buy and then the `balanceOf` the callers address for this contract is increased by the amount of tokens. Furthermore, to be able to sell a token the caller must have the number of tokens in the balance stored on the contract and then the balance is amended and the funds are released to the caller.

To see where the vulnerability might occur, consider what happens when the following function is called `buy{value: 0}(2**255)`, where the buy function is called with `numTokens` of 2^{255} and no ether is sent to the function. The buy will pass and amend the `balanceOf` for the caller to 2^{255} without them having sent any ether to the contract. They will now be able to call the `sell` function and be able to withdraw ether without having ever sent ether.

Common Weakness Enumeration[2] which is a registry of common weaknesses present in software ranks the likelihood of exploiting this weakness as a medium.

2.2.2 Reentrancy

Along with vulnerabilities that are shared among most programming environments like integer overflow, Ethereum smart contracts also come with domain specific vulnerabilities that occur as a result of the programming and execution interface. Many security vulnerability reviews and papers[37, 38, 8] on tools that detect security vulnerabilities have all chosen to note the reentrancy attack, and rightfully so as it allows the attacker to siphon off the entire balance of the contract.

To explain the reentrancy attack one must be made aware of the API for smart contracts. Smart contracts are able to send and recieve ether from or to other addresses may they be user wallets or other smart contracts. Smart contracts are able

```
pragma solidity ^0.4.21;
contract TokenSale {
    mapping(address => uint256) public balanceOf;
    uint256 constant PRICE_PER_TOKEN = 1 ether;

    function buy(uint256 numTokens) public payable {
        require(msg.value == numTokens * PRICE_PER_TOKEN);

        balanceOf[msg.sender] += numTokens;
    }

    function sell(uint256 numTokens) public {
        require(balanceOf[msg.sender] >= numTokens);

        balanceOf[msg.sender] -= numTokens;
        msg.sender.transfer(numTokens * PRICE_PER_TOKEN);
    }
}
```

Listing 2.2: A Token Sale contract with an overflow vulnerability

	send	transfer	call
Gas Cost	2300 wei	2300 wei	all gas or specified amount
Return type	bool	throws error	bool

Table 2.1: List of differences between send, transfer and call to send ether to an address from a smart contract

to send ether in multiple ways, either by using the `reciever.send(amount)` call, `reciever.transfer(amount)` or the `reciever.call{value: amount}("")` where reciever is of the address type and the amount is in wei with 1 ether is 10^{18} wei[2]. Table 2.1 illustrates the differences between the different methods. It it no longer recommended to use `send` or `transfer` due to the fixed gas cost of the function call because it makes the contract vulnerable to gas cost changes, the recommended way to send ether to other addresses.

When a call has been made to send ether to a contract then the EVM looks for a receive method on the receiving contract if there is no data (parameters) sent along with the call or then it looks for a fallback method[43]. The full flow chart for what the EVM calls can be found in Figure 2.2. Once the execution has reached the fallback or the receive function the statements in the function body are executed and then the control is passed backed to the caller function.

[2]https://docs.soliditylang.org/en/v0.8.15/units-and-global-variables.html#ether-units

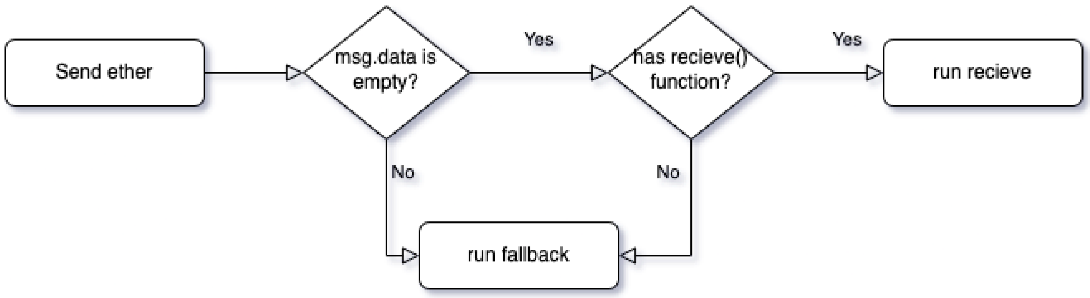


Figure 2.2: Flowchart that illustrates function calls on receiving ether

```
pragma solidity 0.4.24;
contract SimpleDAO {
  mapping (address => uint) public credit;

  function donate(address to) payable public{
    credit[to] += msg.value;
  }

  function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)(""));
      credit[msg.sender]-=amount;
    }
  }

  function queryCredit(address to) view public returns(uint){
    return credit[to];
  }
}
```

Listing 2.3: A Simple dao contract with a reentrancy vulnerability

```
pragma solidity 0.4.24;
contract SimpleDAOAttack {
  SimpleDAO public victim;

  function SimpleDAOAttack(address payable victim_addr) public {
    victim = SimpleDAO(victim_addr);
  }

  function attack() payable {
    require(msg.value >= 1 ether)
    victim.donate{value: 1 ether}(address(this));
    victim.withdraw{value: 0}(1 ether);
  }

  fallback() payable {
    if (address(victim).balance >= 1 ether) {
      victim.withdraw{value: 0}(1 ether);
    }
  }
}
```

Listing 2.4: An attack contract that exploits the reentrancy vulnerability in Listing 2.3

However, let us consider the contract in Listing 2.3(SWC-107). It is a simple contract that allows users to donate to the contract to increase their credit, withdraw from the contract to reclaim their donated amount. This contract contains a reentrancy vulnerability and an attack contract can be found in Listing 2.4 which illustrates an exploit constructed for that vulnerability.

Let's consider the attack contract and how it exploits the reentrancy vulnerability. To begin with, the attack contract initialises by calling the `constructor` and setting the victim address then the `attack` function has two successive function calls to the victim contract, `donate` with value of the call and the address of the attack contract and `withdraw` with a 0 value and the function parameter of the value that was donated in the previous call. The call to `withdraw` triggers the `withdraw` function on the original (victim) contract which in turn checks if the withdrawal amount is valid and then send the the amount to the caller contract which here is the attack contract. Now, recall what happens when ether is sent to a contract it goes through the flow shown in Figure 2.2 which states that in this case the `fallback` function is called on the ether receiving contract. However, looking through the `fallback` function on the receiving (attack) contract, it calls the `withdraw` on the victim contract and this causes a recursive loop that keeps taking funds from the victim contract until it no longer has any funds left. This flow is illustrated in the Figure 2.3.

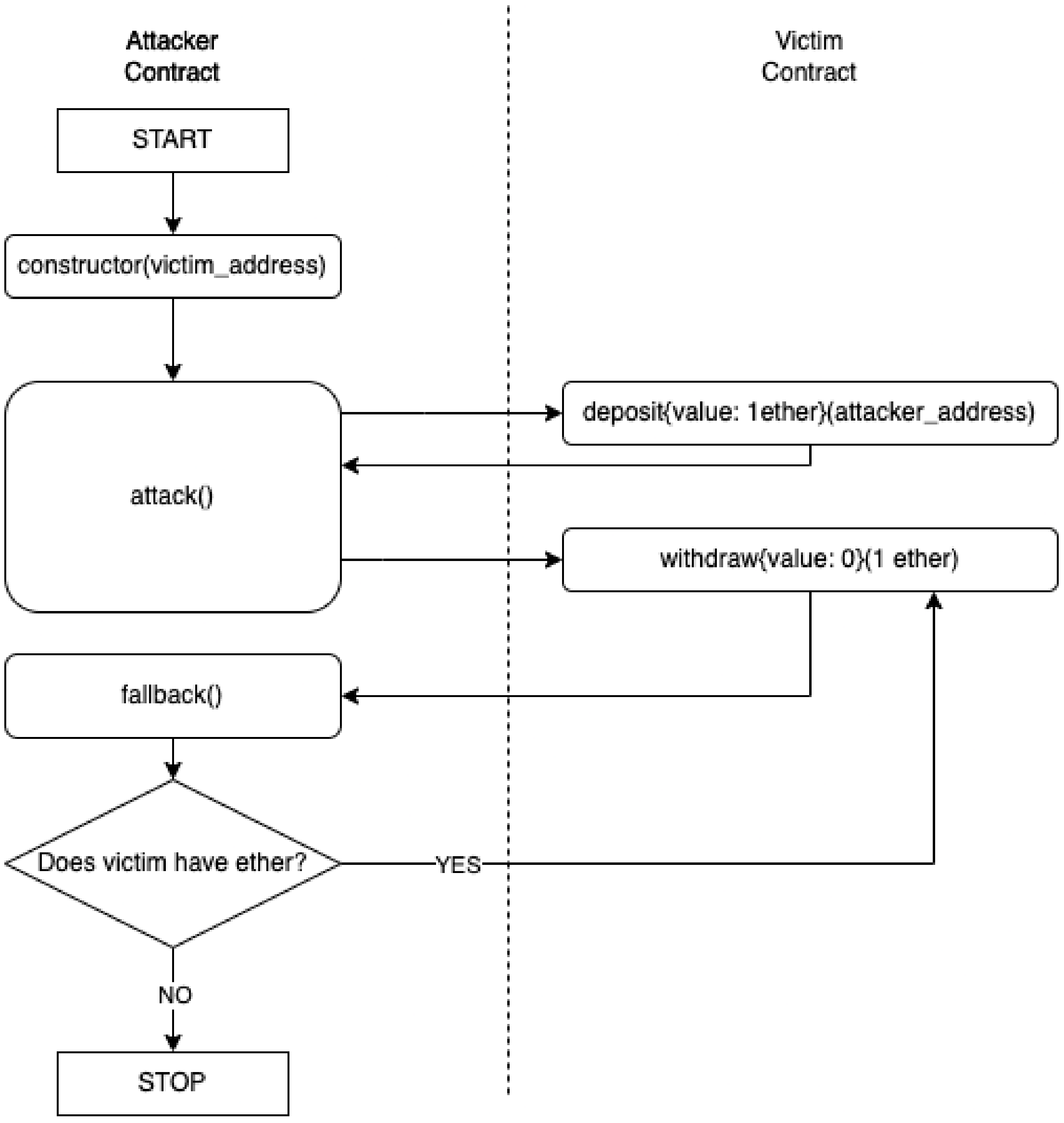

Attacker Contract
Victim Contract
START
constructor(victim_address)
attack()
deposit{value: 1ether}(attacker_address)
withdraw{value: 0}(1 ether)
fallback()
Does victim have ether?
YES
NO
STOP

Figure 2.3: Flow for the reentrancy attack as performed by Listing 2.4

2.2.3 Authorization through tx.origin

Solidity as well as other Ethereum smart contract languages come with a list of pre-populated global variables[3] that that can be used in the body of functions in the smart contract. These globals provide an interface to the programmer to access transaction specific values like the value of ether sent in the contract, the address of the caller of the function, the number of the block that this transaction has been put in and so on. These globals have already been made use of in Listing 2.2, namely the msg global which stores the value of ether sent to the transaction as well as the direct caller of the transaction.

To illustrate the vulnerability one must understand the difference in values stored in the tx.origin and msg.sender variables. As mentioned in the Solidity documentation tx.origin stores the address for ”sender of the transaction (full call chain)”[4] where as msg.sender stores the address for ”sender of the message (current call)”[4]. Each transaction is a series of calls that start from a user on the system that makes the initial call to a smart contract function then that function is free to call whatever other functions it may call. This means that to unambiguously fetch the caller of the function one must mention if they want the caller that started the chain of function calls or simply the caller of the function in question, which is represented by tx.origin and msg.sender, respectively.

This vulnerability arises when these values are interchanged or it is not detected that the values have been interchanged. A phishing exploit is used to illustrate this vulnerability in Listing 2.6 and Listing 2.5, both listings have been adapted from the Sigma Prime security blog[29]. To exploit the vulnerability, the attacker will convince the owner of the contract in Listing 2.6 to send some ether to the address that hosts the attackers contract in Listing 2.5, this may be achieved either by various social engineering methods like imitating an authorized entity.

Once the owner of the victim contract has been convinced to send ether to the attack contract then the fallback on the attack contract will be triggered which will attempt to run the withdrawAll method on the victim contract which is only intended to be run by the owner of victim contract, however, as the contract uses tx.origin to check for who is calling the function it will allow the attacking contract to be authorized and will allow the attacker to withdraw all the funds to an address of their choosing.

[3] `https://docs.soliditylang.org/en/v0.8.16/units-and-global-variables.html#special-variables-and-functions`

[4] `https://docs.soliditylang.org/en/v0.8.16/units-and-global-variables.html#special-variables-and-functions`

```
contract AttackContract {

    Victim victimContract;
    address attacker; // The attackers address to receive funds.

    constructor (Victim _victimContract, address _attackerAddress) {
        victimContract = _victimContract;
        attacker = _attackerAddress;
    }

    function () payable {
        victimContract.withdrawAll(attacker);
    }
}
```

Listing 2.5: Victim contract for the tx.origin vulnerability

```
contract Victim {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Listing 2.6: Attack contract for the tx.origin vulnerability

2.3 SMT Based bound model checking

2.3.1 Bounded Model Checking

Model checking[9] is an automatic formal verification method that allows for the checking of the correctness of a system with respect to a given specification where the system has been modelled as a finite state machine. A model can be represented as a directed graph where the vertices act as states of the system and edges represent the possible transitions between the states. Model checking is a verification method where the state space is explored to look for possible paths (or lack of) that would violate properties outlined by a specification.

Model checking often makes use of two very useful components Kripke structure and Linear temporal logic[15]. The first defines a abstract structure that allows for the models to be represented as finite state machines and the latter is a logic that aids the definition of the specification and the properties within. A Kripke structure, M, is a 4-tuple $M = (S, I, T, L)$ where S is a set of states, I is the set of initial states, T is a set of transitions from a state in S to another state in S which are represented as a 2-tuple, and lastly L is a function that takes a state in S to a set of atomic propositions. One can note that M is simply a more algebraic representation of the graph representation of a model. A path π in M is defined as a sequence of states $s_0, s_1, \ldots$ such that each successive state in the path has a valid transition in T.

LTL is an extension of classical logic that deals with paths in terms of the fact that it carries over the boolean operators from classical logic and it also add multiple operators that allow the semantics to be extended to paths and into the temporal domain. We will for the sake of simplicity consider a fragment of this logic that is PLTL (propositional linear temporal logic) which deals with only propositional formulas. To aid the semantics in Figure 2.4, here is some notation, Let π be a path:

- $\pi(i)$ is the i-th state in the path
- π_i is the path π starting at the i-th state in the path

Basic model checking, however, suffered from the problem of state explosion[9] for real systems where lots of variables are in play and the number of states expand exponentially. This lead to advances in the field and discovery of many other techniques like symbolic model checking and bounded model checking. Bounded model checking worked by checking finite paths upto some length k for a path that incurred a property violation and if no violation is found then the value of k is incremented by

$\pi \models p$	iff	$p \in L(\pi(0))$
$\pi \models \neg f$	iff	$\pi \not\models f$
$\pi \models f \wedge g$	iff	$\pi \models f$ and $\pi \models g$
$\pi \models \mathbf{X}f$	iff	$\pi_1 \models f$
$\pi \models \mathbf{G}f$	iff	$\pi_i \models f$ for all $i \geq 0$
$\pi \models \mathbf{F}f$	iff	$\pi_i \models f$ for some $i \geq 0$
$\pi \models f\mathbf{U}g$	iff	$\pi_i \models g$ for some $i \geq 0$ and $\pi_j \models f$ for all $0 \leq j < i$
$\pi \models f\mathbf{R}g$	iff	$\pi_i \models g$ if for all $j < i$, $\pi_j \not\models f$

Figure 2.4: LTL semantics defining when f holds on π

1 and rechecks for property violations. This repeats until either a violation is found, some intractibility bound is surpassed (eg. timeout) or a pre-known upper bound is reached[15, 17].

Given a Kripke structure M and a formula f which encodes a property for which the correctness is to be checked and a bound k, bound model checking on the triple (M, f, k) can be reduced to a propositional formula for which the satistfiability/validity can be checked for by making use of SAT solvers. This is achieved by first constraining the path to be a valid path, then defining a loop condition for paths with loops and then finally transforming to the propositional formula, details of the reduction can be found in [15].

As programs and procedures can also be represented as a finite state machine, bounded model checking techniques has been leveraged to validate and verify programs for properties like deadlock, integer overflow and such giving certainty to the author of the program that the code is safe from the vulnerabilities it has been checked for (upto a certain bound of loop unrolling).

2.3.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of finding if a first order formula is satisfiable or not relative to axioms defined from a set of theories like the theory of arrays and equality. The advent of SMT solvers allow for a large variety of first order formulas now that they are aided by other theories. Some important theories in this case is LIA (Linear integer arithmetic) which allows for linear arithmetic in integers and contraint solving on integer decision variables, UF (Uninterpreted functions), Theory of arrays which defines axioms on how infinite length arrays will behave. SMT allows for using more than boolean variables which further enables its utility in solving a plethora of problems.

SMT allows us to extend the Bound model checking into the world of first order formulae. This extension cements it utility when it comes to verifying programs which often have arithmetic and array operations. The theories and reasiong methods along those theories make SMT solvers a formidible force when it comes to finding property violations in models.

SMT solver employ complex and locally optimised techniques one of which is known as the lazy approach[10] and is implemented in a majority of the cases. This approach utilises a hybrid approach and the efficiency that SAT solvers have been able to achieve to prune out the possible values for a satisfiable decision by the SMT. The approach is as follows with an input quantifier free formula F:

1. Abstract atoms from F to propositional variables to construct a propositional formula F'
2. Use SAT solver to find model for F'
3. If unsat then return unsatisfiable
4. Using the abstraction and the propositional model construct a clause g with the atoms corresponding to the model in the abstraction
5. Check sat of g in SMT (wrt the given theories)
6. If satisfiable then return satisfiable
7. Assign $F := F \land \neg g$
8. Goto 1
9. END

2.4 ESBMC

ESBMC[19] is an SMT based bounded model checker that as of writing has complete support for C programs including concurrency and partial support for other languages like Kotlin[24], C++ and Solidity[39, 40]. ESBMC is amongst the more popular bound model checkers winning awards numerous times in multiple SV-COMP categories[11, 12, 13]. It is a fork of CBMC[18] which was amongst the early bound model checkers utilizing SAT solvers to verify properties in software.

The current pipeline of ESBMC is split up into three main parts the frontend, middle end and the backend. The front end is a language specific collection of programs which developed for different languages that ESBMC aims to verify. It is responsible for performing lexical analysis and parsing the source file as well as performing type checking to ensure the program is correctly typed and also converting the source code into an internal representation that represents the semantics of the code in the source file.

The middle end is a part of the software which is language independent and is responsible for producing the safety properties as well the model encoding of the code into a logical formula. It does so by converting the internal representation to a SSA form without any loops where loops are replaced with conditionals and goto statements and then converting that to logical formulae that are in the form $C \wedge \neg P$ where C represents the encoding of the source code and P represents the encoding of the specification properties that are being checked for.

Finally, the backend is the powerhouse of the software which is responsible for the actual reasoning part of the software where the logical formulae are checked for satisfiability with the aid of SMT solver, of which ESBMC supports quite a few, and then if the formula is unsatisfiable then return back to to the user that no property violations were found upto a bound (user provided) k or if a model is found for the formula then ESBMC interprets the model and returns back to the user a counterexample state in the source program that would yeild in a property violation. Figure 2.5 illustrates the pipeline for ESBMC for C programs[19].

2.5 Programming Language Theory and AST

This project revolves around static analysis which is analysing programs and reasoning about property violations without running the program. This requires an understanding of how programming languages are defined and how their flow can be extracted from what is simply string of source code. This understanding is a partial fragment of the knowledge of how compilers convert a string of source code to an executable, namely, the front end of a compiler.

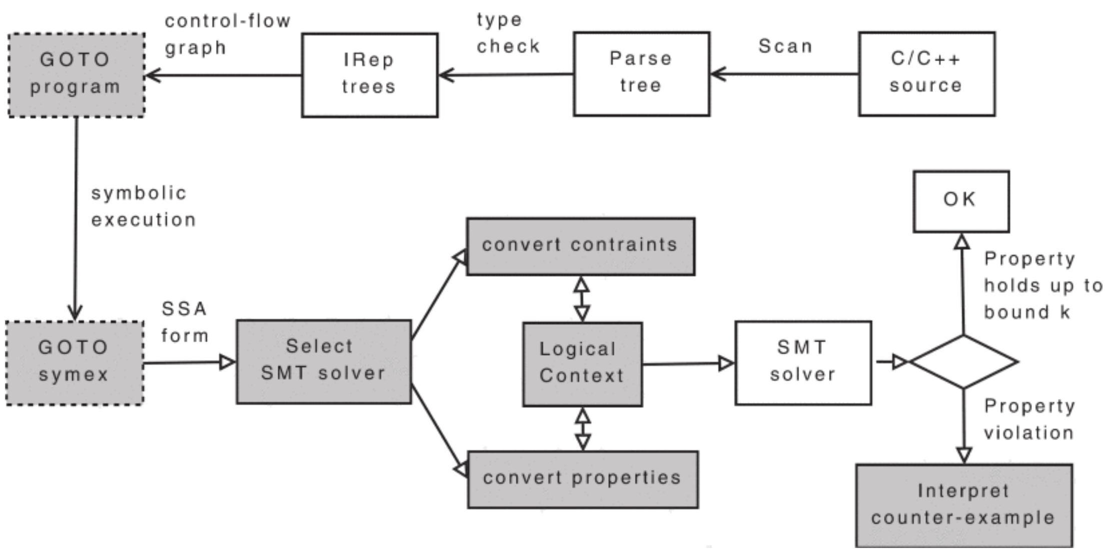


Figure 2.5: Pipeline for C program verification in ESBMC

All programming languages are defined on the basis of a grammar[22], it is a set of rules that defines in clear terms what is a valid string of source code and what is not a valid string of source code. A grammar can be though of as a tuple, (N,S,T,P), of non terminal symbols N, an initial symbol S, terminal symbol T and a set of production rules P that take non terminal symbols to an ordered pair of non terminal and terminal symbols.

By using these production rules, one can push the source code through them and obtain what is known as an Abstract Syntax Tree which is a tree data structure that depicts the parsing of the string and what production rules were applied to get a complete parsing of the source code. Each node stores either a terminal or non terminal symbol and the children are the elements of the tuple in the production rule that was applied. All leaves of the tree are terminal symbols. The advantage to having the source code in this format is that this allows a structured traversal through the code while preserving the semantic information in the code.

2.6 Related work

Verification and the use of automated verification methods is not novel to smart contracts as there are a multitude of already existing works that claim to be able to detect smart contract vulnerabilities of which we will consider 6: Remix IDE[4], Mythril[35], Smartcheck[42], Slither[23], and ESBMC Solidity[39, 40].

2.6.1 Remix IDE

Remix is one of the most popular IDEs out there to develop smart contracts. It is a browser based IDE an comes with a plethora of features that aid blockchain developers like deploying to the blockchain and compilling with diferent compiler version of solidity. This deep integration with the blockchain operations has made block chain developers flock to Remix when choosing an IDE to develop smart contracts. Moreover, it is also equipped with a static analysis tool that is capable of detecting many well known bugs as well as security vulnerabilities, it also highlights bad practices and can be configured to look for only specific vulnerabilities. It operates directly on the source code and consists mostly of pattern matching techniques.

2.6.2 Mythril

Mythril is a static analysis tool hosted by ConsenSys and available open sourced. It operates on the EVM bytecode that is produced after compilation of a smart contract. This makes it much more robust as it is language independant and can also verify programs that exist on other chains that also use EVM as its runtime environment. It makes use of much more sophisticated techniques than Remix like symbolic execution, SMT based taint analysis.

2.6.3 Smartcheck

Smartcheck is another static analysis tool but its methods are simpler and more straight forward. It converts Solidity source code into XML based intermediate representation and the performs XPath queries on the intermediate representation to detect vulnerabilities. In essence it employs pattern matching to detect vulnerabilities in smart contracts. It maintains a list of XPath queries which are matching criteria and iterative matches the IR to each of the queries, if a match is found then reports the match.

2.6.4 Slither

Slither is a static analysis framework that works on solidity source files. Contrary to the other tools on the list it doesn't only look for security vulnerabilities but attempts to detect automatically areas of optimization in the code. It does so by using dataflow and taint tracking which are both commonly used program analisys methods. It operates on the SOlidity source code and converts it to its own internal representation SlithIR which is a SSA form with a reduced instruction set to aid decreasing the complexity of analysis

2.6.5 ESBMC-Solidity

Amongst the most recent additions to the ESBMC family is ESBMC-Solidity[39, 40] which extends the verification prowess of ESBMC to the realm of smart contracts. This implementation for the verification of Ethereum smart contracts written in Solidity is a proof of concept and an exploratory endeavour to judge if it is worthwhile to verify Solidity contracts with ESBMC. The literature that provides the details on its implementation as well as the evaluation data against other players in the same field of smart contract verification show that ESBMC is a force to be reckoned with as it is able to detect and provide counter examples for property violations that some of the most popular tools in the industry are unable to do[39]. It operates on the AST JSON produced by the Solidity compiler and converts that to the ESBMC internal representation which is then passed to the middle end of ESBMC where the IR is converted to assertions and the property violations are checked for.

Currently, before the start of this project, ESBMC Solidity only works on single runs of functions with support for only one type of variable (uint8) and has no support for structs, constructors, or any of the solidity globals. These missing feature do not severely limit the expressibility of the partial grammar that is implemented by much but do make it difficult to be used to verify actual smart contracts.

Moreover, the current implementation of ESBMC Solidity has a very basic pattern matching check for the tx.origin vulnerability discussed in Section 2.2.3 which simply scans the first level of statements and checks if it contains the identifier tx.origin. As for the integer overflows, it is able to detect the overflows but not with the developer annotating the operations with assert statements to check for overflow. Lastly, it has no detection ability to detect Reentrancy bugs which have been the cause of lots of pain with the smart contract community.

Chapter 3

Methodology & Implementation

This chapter outlines the details of the implementation of the project as well as the design considerations and their respective analyses. The project aims to achieve two main objectives, expand the supported grammar for ESBMC Solidity and enable ESBMC Solidity to detect vulnerabilities in smart contracts. We will discuss the design challenges and decisions that were made in the process of achieving the objectives. First we will discuss the current implementation of ESBMC Solidity and the path forward to extending or improving that implementation.

3.1 Opcode Implementation

Currently, ESBMC Solidity performs validation once on a single function within a single contract. It employs a grammar based technique where it traverses through the AST of the Solidity source file and then populates the internal representation objects with data from the AST like the declaration of variables and structs, and the semantic flow of the program. It does so by drawing likeness to C based constructs and imtating the conversion of the C grammar construct to the ESBMC IR.

The trouble here is, that the Solidity grammar and feature set doesn't completely align with the C grammar and feature set which means that for certain grammar constructs the implementation may not have the semantics in the IR to be able to handle the grammar construct. Some examples of this include, associative arrays, classes and exceptions. The lack of these constructs in the IR mean that instead of being able to extend simply the front end for solidity in ESBMC, one would have to add types, semantics, and even modify the current GOTO language to be able to represent these constructs.

In the interest of efficiency and a more complete verification of the language without having to modify the middle end of ESBMC and dealing with the breaking changes that that might bring for other front ends, we chose to explore an opcode based implementation for verification. Every smart contract is compiled to down to bytecode to be executed in the run-time environment of the EVM. The EVM is a stack based machine which means it maintains an internal stack data structure which is somewhat analogous to registers in register based machines, also, each value put on the stack is 32 bytes. The run-time also houses a notion of storage and memory variables where the storage variables are a persistent store and the memory variables are an ephemeral store, both the storage and memory are random access data structures. They would be used to store the addresses or values of local variables as well as state variables. The bytecode is made up of a string of values which is a string of bytes where each byte is an opcode or a parameter to an opcode. These opcodes are hard coded into the EVM and each one has a operation definition as to what is done when that particular opcode is encountered by the EVM. For example when the ADD opcode is encountered two values are popped of the stack added together and then pushed back onto the stack.

This stack based architecture for the EVM is quite similar to the implementation of JVM which also works as a stack based machine and operates on opcodes. This lead to the exploration of a verification tool that was an extension to CBMC to verify Java programs known as JBMC. It operated on the Java opcode produced by the compiler and performed static analysis on the Java opcodes to produce verification of the source code. A similar approach to JBMC was considered here for Solidity based smart contracts which compiles the smart contract into EVM byte-code and then builds a front-end around the byte-codes.

However, this method raises another challenge which is that EVM bytecode strips away all notions of type, which means that checking for operations like overflow are no longer possible as we do not know the size of the integer value to begin with as all values on the stack are 32 bytes. This could however be mitigated by adding metadata to the values on the stack based on the opcode that is used to push the value on the stack, like PUSH1 indicates the value is a single byte, PUSH2 indicates that the value is 2 bytes.

This technique although, cannot be guaranteed to work as the compiler is always looking to perform optimizations to lower the gas cost to run a function as well as improve the run time for the function. There are two in particular that may cause issues, namely, constant folding and storage call optimization. Constant folding propagates a

Advantages	Disadvantages
Independence from implementation language of smart contract	Lack of debugging metadata line variable names and source locations.
Structured and short list of opcodes	Compiler optimizations make it very difficult to perform counterexample generation
	Rebuilding the current Solidity verification on ESBMC from scratch

Table 3.1: Table of advantages and disadvantages of implementing verification by opcode

non changing variable through all its references and eliminates the declaration of the variable altogether. This poses as issue as this will abstract away any integer overflows in the case that the overflow has happened on a constant value.

To explain storage call optimization, one must understand that every opcode comes with an associated gas cost which is a fee (paid by the caller) that compensates the miner for executing the function and updating the state. Calls to update the values stored in storage are made by the SSTORE opcode and can costs a hefty gas fee of 22100 compared to a write is memory which costs about 3 gas plus the gas to expand the memory (low). This makes it worthwhile to limit the amount of writes to storage by combining different write values, for example if we wanted to write `0x01` to position 0 on the storage and `0x05` at position 1, it would be much cheaper to make a single write of the value `0x0105` to storage. This optimization causes a change in how the variables are declared and maintained which means when property violations are found on storage values we cannot keep track of storage values for individual variables as we cannot differentiate between a combined write of `0x0105` representing two separate updates and an uncombined write of `0x0105` to update a single variable. One may ask the property violations should be found regardless as the compiler will also encode this combined writing and reading for other references of the variable like in assert statements which is true but without the additional data for what variable has been updated we will be unable to construct the counter example for the user when a violation is found.

Based on the limitations of the opcode approach to verification, we find it more acceptable to have a smaller coverage over the grammar but allow for the full power of ESBMC to be utilised on fragment of grammar that is supported which delivers counter example generation.

3.2 Expanding supported grammar

To be able to extend the currently implemented it is important to understand the current implementation in detail. The objective of the ESBMC solidity module is to convert from the AST json nodes that are provided by the Solidity compiler to the internal representation which is known across the repository as *irept*. The objective is to construct a symbol table which will then be used to construct a CFG later for generating a first order formulas.

irept is the super class for more specific internal classes like *exprt* which stores the semantics of an expression, *symbolt* which stores the data required for a symbol in a symbol table and so on. The entry point for this conversion is the *convert* function under the *solidity_convertert* class. It looks for the top level *ContractDefinition* node and then performs a pattern based verification on the node to check for tx.origin occurrences, thereafter, it iterates through the children of the contract definition which are assumed to be declarations of some sort and construct and add the symbol corresponding to the declaration to the context/symbol table.

3.2.1 Integer types

The first natural extension was to increase the scope of variable types that ESBMC Solidity supports. Solidity has two main types of integers, signed and unsigned that range in their bit widths from 8 to 256 bits. The current implementation only has support for the unsigned 8 bit integer and this implementation is leveraging helper classes from the C frontend to be able to create the appropriate types.

As most of the code base assumes only one possible elementary type there are multiple items to consider when expanding the number of types for variables. Firstly, as there is only one expected type this means that all integer literals are assumed to be of that type as well. Secondly, once variables can have multiple types we need to consider promotion of variables to higher bit types or demotion to lower bit types when arithmetic is performed between two variables of the same signed-ness but different bit widths. Thirdly, we need to consider casting of one type to another and obeying the semantics of these casts.

Firstly, we must add the recognition of all these new integer types in the AST

```
{
    "typeDescriptions": {
        "typeIdentifier": "t_int8",
        "typeString": "int8"
    }
}
```

Listing 3.1: Example of a type descriptions object

traversal. The point of interest is when the traversal has reached a declaration node for a variable, this is the point when the type of the variable is queried to figure out what internal type must be constructed for the variable in focus. Type information for a variable in the AST json lives in the value of a key known as ”typeDescriptions” which contains a ”typeIdentifier” and ”typeString” key, the latter contains a human friendly readable string and the latter contains a more explicit string that details out further type information like the size of an array or if the value is in storage or memory. In our case, the type string key corresponds exactly to the type of the declaration. Listing 3.1 demonstrates an example type descriptions object. As there is a clear pattern in the type string where unsigned and signed integer types start with uint and in respectively, we can pattern match the unsigned types with ”`uint(\d+)`” and the signed types with ”`int(\d+)`” and use the matching groups to be able to recognize the bitwidth of the type.

Next, the type for the integer variable must be created which holds both its signedness and its bitwidth. To construct the uint8 type for a variable the unsigned char type that is also used in the C frontend is used but this misses out on the granularity that Solidity provides. We instead opt to make use of the more granular types signedbv_typet and unsignedbv_typet which simply represent the types as a bitvector of the same length.

Lets address the first challenge, which is the parsing of literals. This is a problem due to the AST in the json format as the json format doesn’t allow to traverse up the tree from a specific node but only downwards. This is an issue because by the type we get to a node on the tree that represents a literal we have no way of finding out what the type of that literal should be. This is necessary because literals are expressed as constant expressions in ESBMC and also arithmetic is not allowed between values of different bitwidth. This means we need to pass on more information on the type of the literal when a literal expression is processed.

Take the expression `uint32 i = a - (b + 1);` where `a` and `b` are of type `uint8`, the

```json
{
  "declarations": [
    {
      "name": "a",
      "nodeType": "VariableDeclaration",
      "typeDescriptions": {
        "typeIdentifier": "t_uint8",
        "typeString": "uint8"
      }
    }
  ],
  "initialValue": {
    "commonType": {
      "typeIdentifier": "t_uint8",
      "typeString": "uint8"
    },
    "leftExpression": {
      "name": "b",
      "nodeType": "Identifier",
      "typeDescriptions": {
        "typeIdentifier": "t_uint8",
        "typeString": "uint8"
      }
    },
    "nodeType": "BinaryOperation",
    "operator": "+",
    "rightExpression": {
      "kind": "number",
      "nodeType": "Literal",
      "typeDescriptions": {
        "typeIdentifier": "t_rational_1_by_1",
        "typeString": "int_const 1"
      },
      "value": "1"
    },
    "typeDescriptions": {
      "typeIdentifier": "t_uint8",
      "typeString": "uint8"
    }
  },
  "nodeType": "VariableDeclarationStatement"
}
```

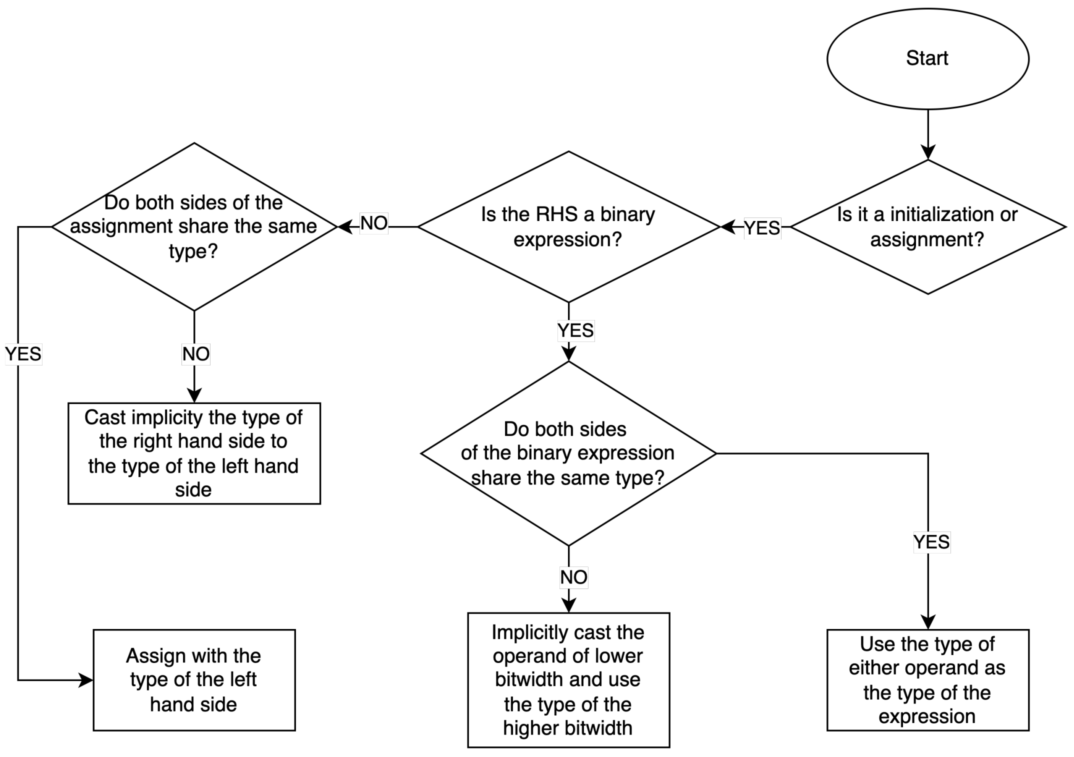
Listing 3.2: Summarized AST for `uint32 i = a - (b + 1);` in json

```
{
  "initialValue": {
    "leftExpression": {
      "name": "b",
      "nodeType": "Identifier",
      "typeDescriptions": {
        "typeIdentifier": "t_uint8",
        "typeString": "uint8"
      }
    },
    "nodeType": "BinaryOperation",
    "operator": "+",
    "rightExpression": {
      "nodeType": "Literal",
      "typeDescriptions": {
        "typeIdentifier": "t_rational_1_by_1",
        "typeString": "int_const 1"
      },
      "metadata_esbmc": {
        "typeDescriptions": {
          "typeIdentifier": "t_uint8",
          "typeString": "uint8"
        }
      },
      "value": "1"
    }
  },
  "nodeType": "VariableDeclarationStatement"
}
```

Listing 3.3: Summarized AST for `uint32 i = a - (b + 1);` in json with metadata

summarized AST json nodes can be seen in Listing 3.2. We can see that the Literal node in the right expression of the + binary operator has no additional type information regarding the expression that it is used in and simply has the type string of ”int_const”. This means during compilation the compiler does additional work to coerce the type of literals based on their usage. So anytime either side of the operator is a literal we pass the common type of the expression to the literal which in this case means we would add metadata to the Literal node that contains the type descriptions information from the parent expression of the Literal node. Listing 3.3 is what the AST json looks after the amendment of the metadata.

Solidity also allows statements where arithmetic operators as well as assignments can happen between different types of certain variety by implicitly converting the types once the operation is performed. Taken from the solidity documentation, ”If an operator is applied to different types, the compiler tries to implicitly convert one of the



Start
Is it a initialization or assignment?
YES
Is the RHS a binary expression?
NO
Do both sides of the assignment share the same type?
YES
NO
Cast implicity the type of the right hand side to the type of the left hand side
Assign with the type of the left hand side
YES
Do both sides of the binary expression share the same type?
NO
Implicitly cast the operand of lower bitwidth and use the type of the higher bitwidth
YES
Use the type of either operand as the type of the expression

Figure 3.1: Flowchart illustrating the implementation of type coercion for expressions

operands to the type of the other (the same is true for assignments). This means that operations are always performed in the type of one of the operands."[1]. This means when deciding the types of expressions it is not enough to consider the type of the expression itself but also the type of the parent. The implementation follows the flow illustrated in Figure 3.1. We inspect on an assignment statement the expression type of its right hand side and then coerce that into type (if possible) of the left hand side of the assignment. This logic also carries over to when we evaluate binary expression for other reasons like index values and function parameters.

3.2.2 Structs

To implement the grammar construct of stucts one must explore the lifetime and usage of structs. Structs are different to other grammar constructs as they are not accompanied by in built types but rather depend on a user defined type. This would require us to dive into the *typet* class which stores the types of symbols in the IR of ESBMC. First let us look at the usage of a struct in a sample Solidity smart contract in Listing

[1]`https://docs.soliditylang.org/en/v0.8.16/types.html#implicit-conversions`

```
pragma solidity ^0.5.0;

contract test {
    struct Foo {
        uint8 a;
        uint16 b;
    }
    function bar() public {
        Foo memory foo1;
        foo1.a = 1;
        foo1.b = 2;
        Foo memory foo2 = Foo(1,2);
        Foo memory foo3 = Foo({b:3, a:4});

    }
}
```

Listing 3.4: An example of struct usage in solidity

3.4. This listing shows the definition of a struct, declaration and initialization of struct variables, assignment of struct values, and member access.

The process of converting a struct into the IR included of the following operations:

- Convert the struct definition
- Convert a struct declaration
- Convert a struct initialization
- Convert member access
- Convert member assignment

First, we convert the struct definition, this requires editing some defaults of ESBMC which assumes that in a ContractDefinition node there can only be either variable declarations of function declarations. An amendment would have to be made to also check for the StructDefinition node. Then to construct the symbol that would hold the type for the struct, first, we construct a unique id based on the name of the struct for the type so we can fetch it back later when a variable is declared of the struct type and create a default symbol that acts as a placeholder for this type with the id constructed above and flag the symbol as being a container for a type. Then, we iterate through the members of the struct in-order of their declaration and construct component objects that hold type information for the members and push them onto the component vector

```
INPUT: init_ast - AST in JSON format
OUTPUT: out - IR expression

struct_type_name = get_struct_name(init_ast.typeDescriptions)
struct_type_id = construct_struct_type_id(struct_type_name)
struct_type_symbol = get_symbol(struct_type_id)

internal_struct_init = get_default_init(struct_type_symbol)

FOR EACH member_value_expr INDEX i IN init_ast.member_values:
    internal_expression = convert(member_value_expr);
    member_type = struct_type_symbol.component_at(i).type
    internal_expression = typecast(internal_expression, member_type)
    internal_struct_init.component_at(i) = internal_expression

out = internal_struct_init
```

Listing 3.5: Pseudo code for generating the IR conversion of a struct initialization

on the type. Components are an abstraction made by the IR to store nested types in type declarations like members in structs or unions.

During declaration, we peek once again at the type string of declaration type and pattern match to look for if it it begins with the word struct which is true for all struct variable declaration nodes. Once we recognize that the type is a struct we then split the type string by spaces and take the second token which is the name of the struct type. We can now use the name of the struct to recreate the id of struct type that was created before and use it to search for the struct type symbol in the IR symbol table. The type from that symbol is then assigned to the variable.

Listing 3.4 depicts multiple ways a struct can be initialized, in both ways it is done by making a function call with the identifier of the struct and passing the member initial values in order or by a named object. The process for converting both of them is quite similar. A high level pseudo code for the conversion is provided in Listing 3.5

Member access is a much easier story, a member access expression is made up of (mainly) 3 things the IR of the base expression, the id of the component on the struct type that corresponds to the member being accessed, and the type of the component that corresponds to the member being accessed. The AST node that represents member access is known as the MemberAccess node in the AST, it contains the name of the member being accessed and expression node that represents the base value ie. the expression on which the member access is performed. We can perform a conversion on the expression to construct the base for the member access IR object. The id can be constructed from the name of the struct which is found on the AST node for the base and the name of the member by the same process that we constructed ids for when the member definitions were populated into the struct type in the symbol table. Lastly, the type of the member comes from the AST as well as the member access comes annotated with a type descriptions key which can be converted to the corresponding type IR object.

3.2.3 Solidity globals

Ethereum smart contract calls contain some metadata regarding the value of ether in the call the gas sent and other out of contract values. This is why Solidity comes with some global variables, namely, block, msg, and tx. They store information regarding the block that the transaction is in, current call metadata like the signature or caller or the value of ether, and details pertaining to the whole transaction like the initiator of the transaction and the gas price of the transaction[2].

These global variables are not an additional feature of the grammar but rather behave like prepopulated variables. This problem is often tackled by verifiers by making use of operational models[33]. To summarize, operational models provide implementations of methods and global variables of usually the standard library but in some cases the run-time environment when the code to be analysed runs some framework. The simplest and most naive way is to append code onto the source file that has implementations of the runtime dependencies of your code. This tactic is most useful when the supported grammar is almost or exactly the same as the language grammar, as the implementations are written in the source language itself the verifier simply considers them as another function or variable declaration and reasons across them as so.

ESBMC also makes use of this technique and before the verification of C programs adds in declarations for a few functions that allow for extra types (like arrays with infinite size) to be available when programs are written as well as verified. In the same spirit we implement a basic operational model that covers some of the globals, before any of the declarations are populated we add symbols to the symbol table for declaration of these globals and leave them uninitialized. ESBMC assumes that uninitialized values can take on any values in the range of its type and treats them as nondeterministic values and reasons about them as taking up any possible value in the range. This behaviour of ESBMC is to our advantage because the globals are not determined by the contract and can take any value in the range of their type.

[2]`https://docs.soliditylang.org/en/v0.8.15/units-and-global-variables.html#block-and-transaction-properties`

```
contract Victim {
    struct __ESBMC_MSG {
        bytes20 sender;
        uint256 value;
    }
    __ESBMC_MSG msg;
    struct __ESBMC_TX {
        bytes20 origin;
        uint256 gasprice;
    }
    __ESBMC_TX tx;
    ...
}
```

Listing 3.6: Listing 2.6 with the operational model prefixed to the contract declarations

One approach as mentioned before is to append the declaration of the globals to the contract under inspection. An example of this can be seen in Listing 3.6. Another approach would be to instead simply add the symbols to symbol table directly without going through the amendment of the declarations to the source file before processing it. ESBMC Solidity operates on the input of a solidity AST, currently it does so by running the compiler on the source file with flags that have the compiler output an AST instead of a binary. Due to having run through the compiler, warning and errors are produced when the code is either not valid according to the grammar or has incorrect semantics or has unrecommended usage. The former approach produces warnings as we are shadowing global variables but as we do not intend to run the contract this warning is acceptable to us. The latter approach adds an obstruction to the maintainability of the code as the operational model information is separate from the AST conversion of the source code. This is why the former approach was preferred.

3.2.4 Constructors

Smart contracts are programs that live on the block chain and can be called upon by users of the chain. On Ethereum when a user has developed their contract ans wishes to publish it to the chain, the EVM runs the constructor on publishing. For contracts that do not have constructors a default constructor is run which is an empty constructor.

ESBMC Solidity has its own start up procedure that defines a flow for the CFG (control flow graph) that is generated from the symbol table. This CFG is represented by the GOTO language which is an internal language that transforms the IR of the code to a semantically equivalent code in the GOTO language with only assignments,

```
pthread_start_main_hook (c:@F@pthread_start_main_hook):
        ...
        // 5 file pthread_lib.c line 126 function
    pthread_start_main_hook
        END_FUNCTION // pthread_start_main_hook

pthread_end_main_hook (c:@F@pthread_end_main_hook):
        ...
        // 9 file pthread_lib.c line 136 function
    pthread_end_main_hook
        END_FUNCTION // pthread_end_main_hook

__ESBMC_sync_fetch_and_add (c:@F@__ESBMC_sync_fetch_and_add):
        ...
        // 17 file builtin_libs.c line 8 function
    __ESBMC_sync_fetch_and_add
        END_FUNCTION // __ESBMC_sync_fetch_and_add

main (c:@F@main):
        // 18 file temp_sol.c line 1 function main
        RETURN: 0
        // 19 file temp_sol.c line 1 function main
        END_FUNCTION // main

foo (foo):
        // 20 file test.sol line 9 function foo
        END_FUNCTION // foo

__ESBMC_main (__ESBMC_main):
        // 21 file esbmc_intrinsics.h line 19
        __ESBMC_alloc=ARRAY_OF(0);
        // 22 file esbmc_intrinsics.h line 22
        __ESBMC_is_dynamic=ARRAY_OF(0);
        // 23 file esbmc_intrinsics.h line 25
        __ESBMC_alloc_size=ARRAY_OF(0);
        // 24 file esbmc_intrinsics.h line 34
        __ESBMC_rounding_mode=0;
        // 25 file pthread_lib.c line 52
        __ESBMC_num_threads_running=0;
        // 26 file pthread_lib.c line 51
        __ESBMC_num_total_threads=0;
        // 27 no location
        FUNCTION_CALL:  pthread_start_main_hook()
        // 28 no location
        FUNCTION_CALL:  foo()
        // 29 no location
        FUNCTION_CALL:  pthread_end_main_hook()
        // 30 file test.sol line 9 function foo
        END_FUNCTION // foo
```

Listing 3.7: GOTO generation from ESBMC for an empty function body

conditionals, and GOTO statements. Listing 3.7 shows the GOTO representation of a contract where the function under analysis is empty. __ESBMC_main is the entry point for the GOTO language. We notice that it has a bunch of variable initializations and then makes 3 function calls. The first and third call is a remnant of the support ESBMC has for concurrency which is implemented in the C frontend.

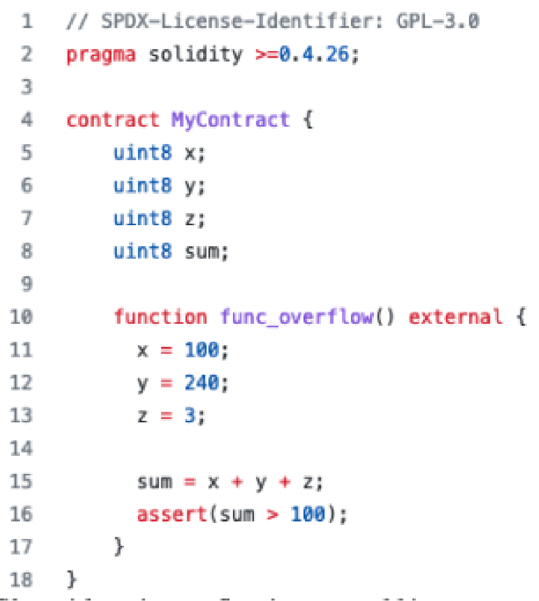
The function of importance is the second call which is the function call to our test function. One can also notice there a main function which is not called anywhere this is because to populate the intrinsics/builtin functions for ESBMC verifier, a conversion is carried out on the C file `int main() { return 0;` this populates the ESBMC intrinsic function and also adds a main function declaration in there. Now it is hard coded into the C frontend that a single function is called between the two pthread function calls, any changes to this would like cause breaking changes in the C frontend verification. Three approaches can be identified here to add the constructor call:

1. Edit the C frontend and add a constructor call to the C frontend between the two pthread calls

2. Wrap the constructor call and the call to the function under inspection in a function call and have the wrapper function be the function called in __ESBMC_main

3. Once the symbol table has been built by the C frontend, iterate the table and find the __ESBMC_main symbol then iterate through its statements and find the statement just before the call to the function in inspection and add the constructor call there

For option 1, the advantage is that it will be the quickest to implement but it brings along stronger coupling between the C frontend and Solidity frontend which is generally frowned upon as these modules should be independent to each other as the two frontends in theory have nothing to do with each other. Option 3 takes away the coupling from the C frontend to the Solidity frontend but does increase the runtime complexity as it performs two linear searches to look for symbols and then performs an insert in the middle of the statements vector which further introduces complexity. Option 2 is a happy compromise between implementation time and maintainability but has the issue that additional guards need to be added so that the wrapper name is not used anywhere in the contract which is an easy fix as we can pick a non-popular identifier and also prevent function names as being that identifier which makes it the preferable option.



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.26;

contract MyContract {
    uint8 x;
    uint8 y;
    uint8 z;
    uint8 sum;

    function func_overflow() external {
      x = 100;
      y = 240;
      z = 3;

      sum = x + y + z;
      assert(sum > 100);
    }
}
```

Figure 3.2: Figure 60 from [39] that illustrates the benchmark for testing overflow

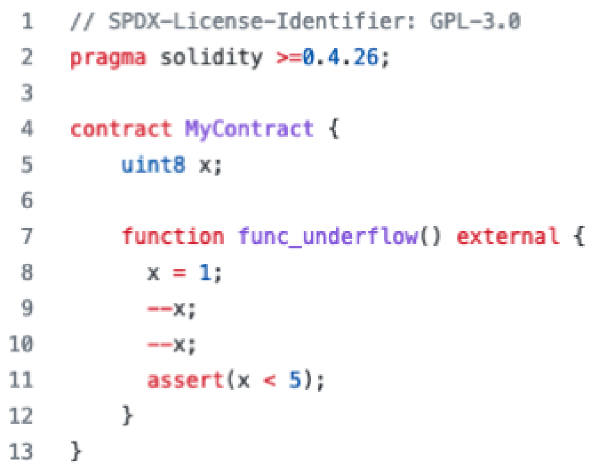
3.3 Detecting Vulnerabilities

The second objective of the this project is to equip ESBMC with better detection abilities for vulnerabilities. The following sections explain the methodology and implementation of detecting complex vulnerabilities within smart contracts.

3.3.1 Fixed bit integer arithmetic

Without any further work done, ESBMC able to detect integer overflow but only with user provided assertions to check for integer overflow. This is made evident by the evaluation criteria presented in the literature for the last work done on ESBMC Solidity[39], on test cases 2 and 3 which are provided in in Figures 3.2 and 3.3. If those test cases are run without the assertions ESBMC returns a VERIFICATION SUCCESSFUL result. This means that implemented feature is not integer overflow detection but rather assertion failure detection. The need for users to add assertions to check for overflow takes away from the automated nature of the vulnerability and rather requires the user to be aware of possible overflows and underflows in the code.

To be able to generate automated assertions for when arithmetic is performed on bit vectors on every arithmetic operation we must add a check for both overflow and underflow. This can be done in a few ways. First, we can look out for binary expressions when traversing the AST and if the binary expression is an arithmetic operation



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.26;

contract MyContract {
    uint8 x;

    function func_underflow() external {
      x = 1;
      --x;
      --x;
      assert(x < 5);
    }
}
```

Figure 3.3: Figure 62 from [39] that illustrates the benchmark for testing underflow

	a >= 0 b >= 0	a <0 b <0	a >= 0 b <0	a <0 b >= 0
+	a+b >a a+b >b	a+b <a a+b <b	no overflow no underflow	no overflow no underflow
-	a-b <= a	no overflow no underflow	a-b >a	no overflow no underflow

Table 3.2: Table with values for assertions for a binary arithmetic operation, `a # b`

we can add an assertion that checks that an overflow does not occur. For example if we encounter a unsigned integer arithmetic expression `a + b` we can add the assertion `assert(a+b >= a && a+b >= b)` for overflow and underflow does not need to be checked for when both values are positive. A flow diagram for the assertions to be added can be found in Table 3.2.

However, as we consider different variations of the grammar we notice that binary expression can have sub expressions and event function calls that return integer values. For sub-expressions, they can have post or pre-increment expressions which means we cannot simply put the same sub-expression in the assert as it is not free of side effects and same goes for function calls. This means we need to extract out sub-expression in an binary expression to some intermediate local variables and replace the values in the binary expression with those local variables and then make our assertions on those local variables.

A keen observation here is that this is the start of converting source code to SSA (single static assignment) form where each variable is assigned to exactly once and each assignment is a simple assignment with no nested expressions. It is possible for

```
...
main (c:@F@main):
        // 19 file test.c line 1 function main
        signed char x;
        // 20 file test.c line 1 function main
        x=127;
        // 21 file test.c line 1 function main
        ASSERT !overflow("+", x, 1) // arithmetic overflow on add
        // 22 file test.c line 1 function main
        x=x + 1;
        // 23 file test.c line 1 function main
        dead x;
        // 24 file test.c line 1 function main
        RETURN: NONDET(signed int)
        // 25 file test.c line 1 function main
        END_FUNCTION // main
...
```

Listing 3.8: GOTO source code excerpt for overflow

us to implement this bug if we refer back to Figure 2.5 we see that this is already being done once the GOTO program is generated from the conversion of the source code to a symbol table with IR objects. Hence it makes more sense to add these assertions during the conversion of the GOTO program to the SMT formulae.

We scour through the different conversions and notice that the SMT has a definition for an overflow guard which means that the GOTO grammar allows for the assertion of an overflow on a value. This is confirmed when we inspect the GOTO conversion for a simple C program that has overflow (Listing 3.8). On further, inspection it seems that the logic for when an overflow assertion is added is dependant on the definition of which overflows are considered undefined by the C standard[26]. This coupling does impact the maintainability of the GOTO converter as the overflow assertion is added not only to on every possible instance of an overflow but rather on every instance of overflow that is undefined by the C standard. The C standard does not define the overflow or underflow of unsigned integer types as undefined which is reflected in the amendment of the overflow assertions for arithmetic operations in the GOTO program.

In the interest of preserving backwards compatibility, no changes were made on the operation of the overflow check on programs but rather an additional check was added that allowed solidity programs to have overflow assertions on both signed and unsigned integer types.

Another matter of discussion, when it comes to overflow detection in ESBMC for smart contracts is the assumption of a single running. We saw in Listing 3.7 that the

```
pragma solidity 0.7.0;
contract MultipleRunOverflow {
    uint8 i = 250;
    function add() public {
        i++;
    }
}
```

Listing 3.9: An example contract with overflow vulnerability on multiple runs of the add function

verification follows the path of calling __ESBMC_main and then calling the function under inspection once. This is okay for C programs because it is correct to assume that the running of the program will begin with the call to main function and no other processes are run before this call. However, in terms of a smart contract we can have functions which do not overflow on their first call but can overflow on a subsequent call. Listing 3.9 illustrates this. A single run of the add function will not overflow but if it is run 10 times then the state variable will overflow.

We already have a wrapper around the main call which calls the constructor and then the function under inspection. We must amend the structure of this function to be able to detect overflows that occur over multiple calls by putting the function call for the function under inspection inside a loop. Reasoning over a loop is one of the key advantages when using a bound model checker, it works by unrolling the loop for each iteration of the loop until a violation is found, the loop is completely unrolled or a threshold of unrolling (user provided) is reached. We will leverage this feature of ESBMC to help us reason that the contract function is safe from overflow even under multiple runs of the function.

3.3.2 Reentrancy

To aid the understanding of automatically detecting a reentrancy vulnerability, we will first discuss how a human verifier would go about verifying a contract to detect a reentrancy vulnerability. Listing 3.10 depicts a simple contract that is vulnerable to the reentracy attack. The function under test is the function `withdraw` which makes a simple check to a state variable and then makes an external call to the sender's address with some ether. The starting point for the detection is establishing if the function even has an external call which is the starting point for the attack. Once the external call has been detected, the verifier must then ask if this call instantly made a recursive call to

```
pragma solidity 0.7.0;
contract ReentrancySimple {
    bool hasSent = false;
    function withdraw(uint8 i) public {
        require(i < 100 && !hasSent);
        msg.sender.call.value(i)("");
        hasSent = true;
    }
}
```

Listing 3.10: A very simple reentrancy vulnerable contract contract

this function again would execution reach the external call again. This is checked by carefully inspecting the code and walking through the execution to find if the guards and conditionals encountered can be bypassed and make the execution arrive at the external call statement. If another external call can be made then this function is vulnerable to a reentrancy attack.

To emulate similar behaviour to a human verifier we will implement an algorithm that will perform static analysis on the function body and determine if a reentrancy call is possible. The algorithm will be three part:

1. Construct the path to the external call
2. Construct the path condition formula from the path
3. Check if the path condition is always true

This algorithm does draw inspiration from a popular static analysis method called symbolic execution where the execution of a program is statically stepped through and the at each statement the state is updated symbolically instead of having concrete values and stored as a boolean formula. Then with the power of modern tools like SMT and SAT solvers the boolean formula is evaluated to find if the program has any executions that may result in failed assertions.

Listing 3.11 expresses the pseudo code to perform the first part of the algorithm which is to construct the execution path to the external call. We step through each statement in the function body and check the children of the statement for the external call if it is not found in the statement then we add the statement to the path and then move onto the next statement otherwise we terminate the path construction and return the path.

After we have constructed the path to the external call we then iterate though the path to locate all the conditions that hold for the execution to reach the external call. In

```
INPUT: ast_node, path = []
OUTPUT: path, bool
construct_path(ast_node, path):
    IF ast_node IS external call:
        path = path.append(ast_node)
        EXIT
    IF ast_node IS declaration OR ast_node IS assignment:
        RETURN path.append(ast_node)
    IF ast_node IS conditional_branch:
        then_path = [conditional_branch.condition]
        then_path = construct_path(conditional_branch.then,
    then_path)

        else_path = [!conditional_branch.condition]
        else_path = construct_path(conditional_branch.else,
    else_path)

        RETURN path.append([then_path, else_path])
    IF ast_node IS loop:
        loop_path = [loop.condition]
        loop_path = construct_path(loop.body, loop_path)
        RETURN path.append([loop_path, !loop.condition])
    IF ast_node IS block:
        FOR EACH child_node IN ast_node:
            path = construct_path(child_node, path)
        RETURN path
    RETURN path
```

Listing 3.11: Pseudo code for constructing the path to the external call that is a potential reentry point

the interest of efficiency this is also where we generate the formulae that will be used to check if the path condition can be a false value. To do so we implement a SSA (static single assignment) type of format for the variables in the formulae. To do so every time a variable `i` is declared with value `v` we represent it in the formulae as `i_0 = v` and for every assignment to the variable thereafter we increment the value in the suffix of the variable by 1 and mark that as the current live variable. This is because in boolean formulae the = stands for equality and not assignment therefore to track assignments and refer to the correct values we must index each assignment.

Another thing to notice is, that at branches that do not have a singular path through them the execution state branches into two separate states. Either one of these states can have our reentrancy vulnerability so we run through both branches and calculate the path condition on both one by one also as the branches will share the same state upto the branch we can leverage backtracking the state to the point of the branch and then calculating the path condition there onward for the other branch.

Similarly, loops also require branching as if the condition of the loop is true then it is run otherwise it exits the loop, similar to a branch where the else condition is the rest of path. To manage the path condition generation with loops we will set an upper limit to which we will expand or unroll the loop and then for each unrolling we will add the condition then the body of the loop and then the negation of the loop condition to the list of assertions for checking the path condition.

Lastly once we have arrived at a completed path condition upto the external call, we now want to repeat the same path and check if a second run can occur or not. We can make use of Z3[21], a SMT solver, to do the reasoning for us on solving if the path condition is satisfiable or not. This will be achieved by making use of Z3 C++ API, the SSA like form of state representation will be a list of declarations of variables based on their type. The path condition will a conjunction of all the conditions that need to hold true for execution to reach the state right before the execution of the external call. Let us use Listing 3.10 to show how a run of this algorithm would look like.

First we will generate the path up to the external call which in this case is on line 6. The path will look something like `[StateDeclarationNode (bool hasSent), DeclarationNode (uint8 i), [[Condition(i < 100 && !hasSent), ExternalCallNode], []]]` when we iterate through these nodes to construct our state and path conditions, we will keep a store of the declared variables their types and their current index, this will be used to create the path condition on the first run. In the example the state formulae will come out to be the following,

```
(declare-const state!hasSent0 Bool)
(declare-const local!0!i0 (_ BitVec 8))
(declare-const local!1!i0 (_ BitVec 8))
% ST
(assert (= state!hasSent0 false))
% PC0
(assert (and (bvule local!0!i0 #x64) (not state!hasSent0)))
% PC1
(assert (not (not (and (bvule local!1!i0 #x64) (not state!hasSent0))
    )))
(check-sat)
```

Listing 3.12: SMT formulae for the verification of contract in Listing 3.10

```
pragma solidity 0.7.0;
contract ReentrancySimpleFixed {
    bool hasSent = false;
    function withdraw(uint8 i) public {
        require(i < 100 && !hasSent);
        hasSent = true;
        msg.sender.call.value(i)("");
    }
}
```

Listing 3.13: Fixed version of Listing 3.10

```
ST  := state!hasSent!0 = false
PC0 := local!0!i!0 < 100 && NOT state!hasSent!0
PC1 := local!1!i!0 < 100 && NOT state!hasSent!0
```

Where ST is the state of the variables right before the execution, PC0 is the path condition upto the external call for the first run of the function and PC1 is the path condition where the local variables are renewed with a new set of local variables and the state variables are persisted. The SMT formulae that are constrcuted from this look like Listing 3.12.

A value of unsat would confirm that there is no reentrancy vulnerability as it would confirm that the negation of the last assertion is valid (always true). The last assertion, in plain English translates to checking that for a second run of the function the path condition can never be true and hence the external call cannot be made. A value of sat would tell us that there is a reentrancy vulnerability present and we can also use the model for the satisfiability to parse values for which this vulnerability can be exploited.

Listing 3.13 shows a fixed version of the reentracy vulnerable contract along with Listing 3.14 which yields an unsat result correctly.

```
(declare-const state!hasSent0 Bool)
(declare-const local!0!i0 (_ BitVec 8))
(declare-const local!1!i0 (_ BitVec 8))
(declare-const state!hasSent1 Bool)
% ST
(assert (= state!hasSent0 false))
(assert (= state!hasSent1 true))
% PC0
(assert (and (bvule local!0!i0 #x64) (not state!hasSent0)))
% PC1
(assert (not (not (and (bvule local!1!i0 #x64) (not state!hasSent1))
    )))
(check-sat)
```

Listing 3.14: SMT formulae for the verification of contract in Listing 3.13

3.3.3 Authorization through tx.origin

This detection is the most straight forward of all the detection as it consists only of pattern matching. Currently, ESBMC Solidity only looks at the top level statements and matches for a require statement that has a binary expression with the comparison operator == and the left hand side of this expression is a tx.origin node. This is mostly the crux of the detection for this vulnerability. This is, however, quite a naive implementation as it ignores some key things, like the require call being in the body of an invoked function within the function body or that the require call having more than one boolean condition.

To amend this we will implement three things, a proper traversal through all nodes, a more robust matching criteria, and checking for the vulnerability in function invocations. To improve the traversal we will traverse the entire body of the function and visit all statements where the require function can appear, this can be within loop bodies, conditional branch bodies, and as sub expressions in statements like declaration and assignment. To update the matching criteria we do not simply stop if the require argument is a binary expression that doesn't have an equality comparison operator but rather we traverse the right and left hand side of the expression looking for an equality comparison and then checking both sides of the expression to look for a tx.origin node.

Lastly, to make the check even more robust we will traverse through function bodies of invoked functions. To fetch a function body based on the function invocation requires multiple steps, where the id of the function must be constructed and then it must be searched for AST and then the body needs to be traversed. The search is also not efficient is it Solidity doesn't require that functions are declared before their invocations which means the entire body of the contract definition must be searched for the function declaration. Also, the function can be invoked more than once with the same parameters which will prompt the same body to be inspected through more than once which is again inefficient. A more efficient manner to deal with this would be to simply, search through the bodies of all functions for a pattern match ignoring the function invocations. If any of the functions contain a match we declare the vulnerability as it still makes the contract vulnerable for a call to that function. If no match is found in any function and every individual function is safe from the vulnerability then we can reason that any function invocation will also be safe from the vulnerability.

Chapter 4

Evaluation

This section contains details on how the project implementations were evaluated, against a previous version of the project and also against related work. All executions were carried out on a Dell Inspiron 17 5748 Core i5 8GB 1TB via a remote connection. The test cases serve two objectives, to showcase the expansion of the grammar an that reasoning is working on the extension of the grammar in comparison to the previous version of the project and to showcase the detection capabilities of the project in comparison to both the older version of the project and other smart contract static analysis tools.

4.1 Test Contracts

The following test cases are designed to be able to verify the implementations in the project. For each test case we run the ESBMC binary with the added changes from this project and then use the unix time command[7] to measure how long it too to run the verification. We use the user time on returned from the time command to dull out any noise that occurs due to other system processes. Each test is run 5 times and a median of those values is taken to further eliminate noise from the reading and not skew the times to outliers. Table 4.1 lists out the tests and their descriptions along with the expected verification output for each test.

Category	Listing ID	Description	Expected
Grammar extension	4.1	Valid contract with constructor	Verification Successful
	4.2	Invalid contract with constructor	Verification Failed
	4.3	Valid contract with signed integers	Verification Successful
	4.4	Invalid contract with signed integers	Verification Failed
	4.5	Valid contract with structs	Verification Successful
	4.6	Invalid contract with structs	Verification Failed
Vulnerability detection	4.7	Contract with external call and reentrancy vulnerability	Verification Failed
	4.8	Contract with external call and no reentrancy vulnerability	Verification Successful
	4.9	Contract with signed integer overflow	Verification Failed
	4.10	Contract with signed integer overflow over multiple runs	Verification Failed
	4.11	Contract with int and uint operations without any overflows or underflows	Verification Successful
	4.12	Contract with signed integer underflow	Verification Failed
	4.13	Contract with signed integer overflow over multiple runs	Verification Failed
	4.14	Contract with tx.origin in the body of a function invocation	Verification Failed
	4.15	Contract with tx.origin in the body of a loop	Verification Failed
	4.16	Contract with tx.origin outside a require call	Verification Successful
	4.17	Contract with unsigned integer overflow	Verification Failed
	4.18	Contract with unsigned integer overflow over multiple runs	Verification Failed
	4.19	Contract with unsigned integer underflow	Verification Failed
	4.20	Contract with unsigned integer overflow over multiple runs	Verification Failed

Table 4.1: Table of tests designed to evaluate the implementations

4.1.1 Grammar extension tests

The design of these tests follow a simple pattern each grammar construct implemented has two tests with assertion statements that have to be verified. One test has a failing assertion and one test has a passing assertion this is to ensure that the grammar construct behaves in both cases and is not a trivial verifier that says all contracts are valid.

```
pragma solidity 0.6.0;
contract MyContract {
    uint8 i = 0;
    constructor() public {
        i = 5;
    }
    function test() public {
        assert(i == 5);
    }
}
```

Listing 4.1: Valid contract with constructor

```
pragma solidity 0.6.0;
contract MyContract {
    uint8 i;
    constructor() public {
        i = 5;
    }
    function test() public {
        assert(i == 0);
    }
}
```

Listing 4.2: Invalid contract with constructor

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        int8 i = 2;
        i = 2 * i - 10;
        assert(i == -6);
    }
}
```

Listing 4.3: Valid contract with signed integers

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        int8 i = 2;
        i = 2 * i - 10;
        assert(i == 2);
    }
}
```

Listing 4.4: Invalid contract with signed integers

```
pragma solidity 0.6.0;
contract MyContract {
    struct a {uint8 i;}
    function test() public {
        a memory s = a(1);
        assert(s.i == 1);
    }
}
```

Listing 4.5: Valid contract with structs

```
pragma solidity 0.6.0;
contract MyContract {
    struct a {uint8 i;}
    function test() public {
        a memory s = a(1);
        assert(s.i == 2);
    }
}
```

Listing 4.6: Invalid contract with structs

Listing ID	Time to run (s)
4.1	0.25
4.2	0.22
4.3	0.23
4.4	0.23
4.5	0.24
4.6	0.24

Table 4.2: Table of run time values for grammar expansion tests, green time values represent correct outcome and red indicated incorrect outcome

Table 4.2 shows the results of running these tests on the binary generated by ESBMC after the implementation from this project have been made. Note, the implementations correctly identify the correct verification for the contracts and do it within reasonable time as well. Implementations also seem efficient from the data as more complex grammar constructs do not significantly increase the run time.

4.1.2 Vulnerability tests

```
pragma solidity 0.6.0;
contract MyContract {
    bool b = false;
    function test() public {
        require(!b);
        msg.sender.call.value(100)("");
        b = true;
    }
}
```

Listing 4.7: Contract with external call and reentrancy vulnerability

```
pragma solidity 0.6.0;
contract MyContract {
    bool b = false;
    function test() public {
        require(!b);
        b = true;
        msg.sender.call.value(100)("");
    }
}
```

Listing 4.8: Contract with external call and no reentrancy vulnerability

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        int8 i = 127;
        i++;
    }
}
```

Listing 4.9: Contract with signed integer overflow

```
pragma solidity 0.6.0;
contract MyContract {
    int8 i = 120;
    function test() public {
        i++;
    }
}
```

Listing 4.10: Contract with signed integer overflow over multiple runs

```
pragma solidity 0.6.0;
contract MyContract {
    function test(uint8 i, int8 j) public {
        if (i < 255) {
            i++;
        }
        if (i > 0) {
            i--;
        }
        if (j < 127) {
            j++;
        }
        if (j > -128) {
            j--;
        }
    }
}
```

Listing 4.11: Contract with int and uint operations with- out any overflows or underflows

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        uint8 i = 0;
        i--;
    }
}
```

Listing 4.12: Contract with signed integer underflow

```
pragma solidity 0.6.0;
contract MyContract {
    int8 i = -120;
    function test() public {
        i--;
    }
}
```

Listing 4.13: Contract with signed integer overflow over multiple runs

```
pragma solidity 0.6.0;
contract MyContract {
    address owner;
    function test() public {
        other();
    }

    function other() public {
        require(tx.origin == owner);
    }
}
```

Listing 4.14: Contract with tx.origin in the body of a function invocation

```
pragma solidity 0.6.0;
contract MyContract {
    address owner;
    function test() public {
        uint8 i = 0;
        while (i++ == 0) {
            require(tx.origin == owner);
        }
    }
}
```

Listing 4.15: Contract with tx.origin in the body of a loop

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        tx.origin;
    }
}
```

Listing 4.16: Contract with tx.origin outside a require call

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        uint8 i = 255;
        i++;
    }
}
```

Listing 4.17: Contract with unsigned integer overflow

```
pragma solidity 0.6.0;
contract MyContract {
    uint8 i = 250;
    function test() public {
        i++;
    }
}
```

Listing 4.18: Contract with unsigned integer overflow over multiple runs

```
pragma solidity 0.6.0;
contract MyContract {
    function test() public {
        int8 i = -128;
        i--;
    }
}
```

Listing 4.19: Contract with unsigned integer underflow

```
pragma solidity 0.6.0;
contract MyContract {
    uint8 i = 5;
    function test() public {
        i--;
    }
}
```

Listing 4.20: Contract with unsigned integer overflow over multiple runs

Test Listing ID	Slither	Mythril	Smartcheck	Remix IDE	ESBMC Solidity
4.7	**0.47**	3.75	3.68	N/A	0.49
4.8	0.46	3.76	3.61	N/A	**0.31**
4.9	0.46	3.25	3.19	N/A	**0.22**
4.10	0.45	4.25	3.61	N/A	**0.28**
4.11	0.45	6.1	3.31	N/A	**0.24**
4.12	0.44	3.13	3.25	N/A	**0.25**
4.13	0.45	4.19	3.34	N/A	**0.22**
4.14	0.45	3.58	3.56	N/A	**0.22**
4.15	0.47	3.42	3.48	N/A	**0.24**
4.16	0.46	3.03	3.47	N/A	**0.24**
4.17	0.45	3.08	3.18	N/A	**0.22**
4.18	0.45	4.21	3.35	N/A	**0.23**
4.19	0.44	3.11	3.14	N/A	**0.22**
4.20	0.45	4.25	3.28	N/A	**0.24**

Table 4.3: Table of run time values (in secs) for vulnerability detection tests, green time values represent correct outcome and red indicated incorrect outcome, the bolded time in each row represents the fastest correct verification

4.2 Comparison to other works

Table 4.3 outlines the run times for each of the tools for the vulnerability detection times. We can see from the data that ESBMC solidity outperforms the other tools in all but one test which is the reentrancy detection, however, as the implementation for reentrancy detection is a naive proof of concept implementation without any optimizations there is potential to squeeze out more performance from ESBMC Solidity for that test. We also notice that current market tools do not perform well when it comes to detecting integer overflows and underflows for single or multiple runs. Another noterworthy mention is, that tools like slither and mythril detected many vulnerabilities in the contracts that ESBMC did not recognize, namely, default visibility for state variables, unchecked return value for call function, sending ether to arbitrary users. Also, when the tools were able to detect the vulnerabilities information and detail about the vulnerability was presented and a severity was presented for each detected vulnerability where as ESBMC displays the same level of failure on all vulnerabilities detected.

From the data we can see that Mythril was the worst perming tool on the benchmarks. It was only able to detect vulnerabilities (or lack there of) 3 out of the 14 tests which is the lowest success rate in the tools tested. It also ran the longest on average making it the least performant of the bunch. Next higher up was smartchecker which detected 4 out of 14 vulnerabilities, followed by a tie for 2nd place between slither and remix.

Note, the Remix IDE has no run time values as the IDE lives in the browser and run the checks in the browser, any timing would be plagued by network latency and rendering latency of the browser which would make the times very unreliable and not of much use. Therefore, we have only depicted the outcome of the verification and omitted the runtimes for the verification.

To great pleasure and relief, we see that ESBMC out performs all the tools for the laid out tests (all but one) by quite a margin cementing the ability for ESBMC to serve is a viable tool in smart contracts for security property verification.

4.3 Limitations

The tests designed to benchmark the implementations are not the most representative of real contracts. They make use of a limited grammar of Solidity and are geared towards depicting the difference in performance for different tools while still staying within the supported grammar of ESBMC Solidity. These benchmarks also do not test scalability of contracts all tests span only one file and often no more than 20 lines in total.

All test cases were also designed to be focused around a single vulnerability and the detection of that vulnerability. It is not representative of how the tools would perform given a contract with multiple vulnerabilities present simultaneously.

Chapter 5

Conclusion & Further work

5.1 Conclusion

In conclusion, the project set out to explore and further develop the current abilities of ESBMC to verify smart contracts in the scope of various security vulnerabilities. By the results of the eevaluation one can claim that this was achieves as development on the tool enhanced its ability to verify smart contracts and gave it the ability to verify well known security vulnerabilities in smart contracts. Moreover, the developments were on par and even exceeded the performance as well as detection capability of well established tool on the market.

5.2 Limitations & Further work

The project discusses in detail of the challenges and the design decisions that helped overcome the challenges along with specific implementation details that allowed ESBMC to detect the vulnerabilities. Even though the project seems to result in a deliverable that is performant, it is still very far from complete. The following are some limitation of ESBMC Solidity and should be considered for further work:

- Missing essential types like mapping which is an associative array in Solidity. Can take inspiration from C++ operational model for maps for ESBMC[33]
- No support for polymorphism or even a second contract
- Cannot detect transaction order dependence, where the function can be run after any number of other function runs. ESBMC Solidity assumes the function under

test is the first and only function run.

- Dynamic array without fixed length are not reasoned about. Inspiration can be taken from C++ operational model for vectors for ESBMC[33].
- Extend the builtin globals operational model to builtin functions that allow hashing and performing arithmetic without bounds

Bibliography

[1] Block chain. URL: https://developer.bitcoin.org/devguide/block_chain.html#introduction.

[2] Common weakness enumeration. URL: https://cwe.mitre.org/.

[3] Overview · smart contract weakness classification and test cases. URL: https://swcregistry.io/.

[4] Solidity static analysis. URL: https://remix-ide.readthedocs.io/en/latest/static_analysis.html.

[5] Types. URL: https://docs.soliditylang.org/en/v0.8.16/types.html#integers.

[6] What is ether (eth)? URL: https://ethereum.org/en/eth/.

[7] *time(1) Linux User's Manual*, July 2013.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[10] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

[11] Dirk Beyer. Software verification with validation of results. pages 331–349, 03 2017. doi:10.1007/978-3-662-54580-5_20.

[12] Dirk Beyer. Automatic verification of c and java programs: Sv-comp 2019. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155, Cham, 2019. Springer International Publishing.

[13] Dirk Beyer. Advances in automatic software verification: Sv-comp 2020. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 347–367, Cham, 2020. Springer International Publishing.

[14] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[15] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[16] Checkmate. The week on-chain (week 18, 2021), May 2021. URL: `https://insights.glassnode.com/the-week-on-chain-week-18-2021/`.

[17] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, Jul 2001. `doi:10.1023/A:1011276507260`.

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[19] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011.

[20] Lucas Cordeiro, Daniel Kroening, and Peter Schrammel. Jbmc: Bounded model checking for java bytecode. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 219–223. Springer, 2019.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. volume 4963, pages 337–340, 04 2008. `doi:10.1007/978-3-540-78800-3_24`.

[22] Thain Douglas. Introduction to compilers and language design second edition. pages 35–69, 85–98. Independently published, 2020.

[23] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. *CoRR*, abs/1908.09878, 2019. URL: `http://arxiv.org/abs/1908.09878`, `arXiv:1908.09878`.

[24] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An industrial-strength C model checker. In 33rd *ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)*, pages 888–891, New York, NY, USA, 2018. ACM. `doi:10.1145/3238147.3240481`.

[25] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[26] ISO. *C11 Standard*, 2011. ISO/IEC 9899:2011.

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[28] Spyros Makridakis and Klitos Christodoulou. Blockchain: Current challenges and future prospects/applications. *Future Internet*, 11(12):258, 2019.

[29] Adrian Manning. Solidity security: Comprehensive list of known attack vectors and common anti-patterns, Oct 2018. URL: `https://blog.sigmaprime.io/solidity-security.html#tx-origin-vuln`.

[30] Steve Marx. Capture the ether - token sale. URL: `https://capturetheether.com/challenges/math/token-sale/`.

[31] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.

[32] Mix. These are the top 10 programming languages in blockchain, May 2019. URL: `https://thenextweb.com/news/javascript-programming-java-cryptocurrency`.

[33] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. Model checking c++ programs. *Software Testing, Verification and Reliability*, 32(1):e1793, 2022. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1793`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1793`, `doi:https://doi.org/10.1002/stvr.1793`.

[34] Felipe R Monteiro, Mikhail R Gadelha, and Lucas C Cordeiro. Model checking c++ programs. *Software Testing, Verification and Reliability*, 32(1):e1793, 2022.

[35] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 9:54, 2018.

[36] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[37] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

[38] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020.

[39] Kunjian Song. Smt-based bounded model checking for solidity smart contracts. Master's thesis, The University of Manchester, 2021.

[40] Kunjian Song, Nedas Matulevicius, Eddie B de Lima Filho, and Lucas C Cordeiro. Esbmc-solidity: An smt-based model checker for solidity smart contracts. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 65–69. IEEE, 2022.

[41] William Stallings and William Stallings. *Cryptographic Hash Function*, page 339–365. Pearson, 2017.

[42] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on*

Emerging Trends in Software Engineering for Blockchain, WETSEB '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3194113.3194115`.

[43] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[44] Congcong Ye, Guoqiang Li, Hongming Cai, Yonggen Gu, and Akira Fukuda. Analysis of security in blockchain: Case study in 51%-attack detecting. In *2018 5th International conference on dependable systems and their applications (DSA)*, pages 15–24. IEEE, 2018.