AN ANALYSIS OF ESBMC'S EFFECTIVENESS ON SWC VULNERABILITIES

2023

John Lindon Robinson

Department of Computer Science

Contents

Word Count: 5800

3

List of Tables

List of Figures

Abstract

An Analysis of ESBMC's effectiveness
On SWC Vulnerabilities
John Lindon Robinson
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2023

This university project aims to develop a benchmark for the ESBMC smart contract checker by creating a large set of vulnerable smart contracts. The smart contracts were written with Solidity, a popular programming language for creating smart contracts on the Ethereum blockchain.

The project involved researching common vulnerabilities in smart contracts and implementing them intentionally in the benchmark contracts. The vulnerabilities targeted include reentrancy attacks, integer overflows and underflows, logic flaws, and other common mistakes that can lead to security vulnerabilities.

The benchmark contracts were then tested with ESBMC, a popular software model checker that is capable of verifying the correctness of smart contracts. The results of the tests were analyzed to determine the effectiveness of ESBMC in detecting and preventing vulnerabilities in smart contracts.

The project aims to provide a useful resource for developers and researchers working on smart contract security. By providing a comprehensive set of vulnerable smart contracts, the benchmark can be used to test and compare different smart contract security tools and techniques, ultimately leading to more secure smart contracts and blockchain applications.

Declaration

No portion of the work referred to in this report 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 Lucas Cordeiro for his guidance and support throughout this project. I would also like to thank my family for their support and encouragement. TODO:

Chapter 1

Introduction

1.1   Motivation

The growing popularity and widespread adoption of smart contracts in various industries have sparked significant interest in ensuring their security, reliability, and robustness. Smart contracts, as self-executing contracts with the terms of the agreement directly written into code, are becoming an essential component of blockchain platforms, such as Bitcoin, Ethereum, Polygon or a suite of other Platforms. They enable secure, decentralized, and automated transactions without intermediaries, increasing efficiency and reducing transaction costs. Despite these potential advantages, poorly designed or vulnerable smart contracts can lead to significant losses, as evidenced by high-profile security breaches, such as a hack on The DAO in 2016, which cost its users 50 Million USD, by exploiting a particular vulnerability.

This dissertation aims to study the effectiveness of the ESBMC (Efficient SMT-Based Context-Bounded Model Checker) in identifying vulnerabilities within smart contracts. ESBMC is a powerful and widely-used tool for formal verification, which has shown great potential in detecting software errors, including concurrency-related bugs and vulnerabilities in smart contracts. However, there is a shortage of research on the performance of ESBMC when applied to smart contracts with known vulnerabilities.

By developing a set of intentionally vulnerable smart contracts to serve as a benchmark, this study seeks to bridge this research gap and provide valuable insights into the capabilities and limitations of ESBMC in detecting and mitigating smart contract vulnerabilities. This benchmark suite will not only help assess the efficiency and accuracy of the ESBMC tool but also contribute to improving and enhancing model-checking

techniques for smart contract verification.

Additionally, the findings of this research can assist smart contract developers and security experts in gaining a deeper understanding of potential vulnerabilities and employing more secure coding practices. By evaluating the ESBMC tool's performance on this benchmark, the study also hopes to foster the development of more robust and effective model-checking tools for smart contract verification, ultimately establishing safer and more trustworthy blockchain ecosystems.

1.2 Research Question, Aim and Objectives

1.2.1 Research Question

How effectively is ESBMC detecting and mitigating vulnerabilities within a benchmark suite of intentionally vulnerable smart contracts?

1.2.2 Aim

This study aims to assess the performance of ESBMC in identifying and addressing vulnerabilities in smart contracts found in the SWC Registry [7] by using a benchmark suite of deliberately vulnerable smart contracts.

1.2.3 Objectives

The objectives for this project are as follows:

1. To develop a benchmark suite of vulnerable smart contracts that simulate real-world security flaws and weaknesses.

2. To evaluate the effectiveness of ESBMC in detecting and analyzing the vulnerabilities within the benchmark suite.

3. To investigate the limitations and challenges of using ESBMC for smart contract verification.

4. To provide recommendations for improving the performance of ESBMC and other model-checking tools in detecting and mitigating smart contract vulnerabilities.

Chapter 2

Background

2.1 Blockchain

A blockchain is a decentralized, distributed ledger technology that enables secure, transparent, and tamper-resistant storage of digital records across a network of participants. It consists of a series of blocks, each containing a list of transactions, which are cryptographically linked and secured using cryptographic algorithms. This structure allows for enhanced security and data integrity, as altering the information in one block would require most network participants' consensus and modifying all subsequent blocks.

Although initially developed for supporting cryptocurrencies like Bitcoin, blockchain technology has evolved to accommodate various applications across many industries, such as finance, supply chain, healthcare, and more [10] [6]. Ethereum has emerged as a leading platform for developing and deploying smart contracts among the different blockchain platforms.

2.1.1 Etherum Blockchain

The Ethereum blockchain, launched in 2015 by Vitalik Buterin and his team, was designed to facilitate the creation, management, and execution of decentralized applications (DApps) and smart contracts. Unlike Bitcoin, which is primarily used for transferring digital currency, Ethereum provides a decentralized virtual machine—the Ethereum Virtual Machine (EVM)—which can execute arbitrary Turing-complete code on the blockchain [5]. This feature allows developers to build and deploy more complex and versatile applications on the Ethereum platform.

Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They automatically execute and enforce the contract's terms when predefined conditions are met without the need for intermediaries. This enables secure, decentralized, and automated transactions on the blockchain, leading to increased efficiency and reduced transaction costs. Ethereum's native cryptocurrency, Ether (ETH), is used to pay for the computational resources and transaction fees required to execute smart contracts on the network. A pertinent fact in this research is that once a smart contract is deployed on the Ethereum blockchain, it cannot be modified or removed. This immutability makes smart contracts a highly attractive target for attackers, as they can potentially cause significant financial losses and damage the organization's reputation. For this reason, smart contracts must be developed and deployed securely.

2.1.2   The Solidity Language

Solidity is a high-level, statically-typed, contract-oriented programming language specifically designed for writing smart contracts on the Ethereum blockchain. Created by Dr. Gavin Wood, Christian Reitwiessner, and their team at Ethereum, Solidity is influenced by other programming languages such as JavaScript, Python, and C++, and is designed to target the Ethereum Virtual Machine (EVM). The EVM executes the compiled bytecode of the smart contracts, which is generated from the Solidity source code.

2.1.2.1   Syntax and Structure

Solidity syntax is similar to JavaScript and employs a curly-bracket () notation for defining code blocks. A Solidity smart contract typically starts with a *pragma* directive, which specifies the version of the Solidity compiler required for the source code. This is followed by the contract definition, which includes the contract's state variables, functions, events, and access modifiers.

```
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 private storedData;
```

```
function set(uint256 x) public {
    storedData = x;
}

function get() public view returns (uint256) {
    return storedData;
}

}
```

The example above demonstrates a simple Solidity contract, *SimpleStorage*, which allows users to store and retrieve an unsigned 256-bit integer value. The contract consists of a private state variable, *storedData*, and two public functions, *set()* and *get()*.

2.1.2.2 Data Types and Variables

Solidity supports various data types, including value types (such as integers, booleans, and addresses) and reference types (such as arrays, mappings, and structs). Additionally, Solidity allows for the declaration of user-defined types, such as enums and structs, to create more complex data structures.

2.1.2.3 Functions and Modifiers

Functions in Solidity are similar to functions in other programming languages, defining a reusable block of code that performs a specific task. Functions can be declared as public, private, external, or internal, which determines their visibility and accessibility within the contract and by other contracts. Functions can also be marked as *view* or *pure*, indicating that they do not modify the contract's state and only read or compute data, respectively.

Modifiers can be used to alter the behavior of functions by appending or prepending additional code to the function's body. They are often used to enforce access control, by requiring certain conditions to be met before the function can be executed, such as requiring the sender to be the contract owner.

2.1.2.4   Events and Inheritance

Events are used in Solidity to emit logs that can be monitored by the contract's users, allowing them to be notified of specific occurrences or state changes within the contract. This is particularly useful for creating event-driven applications and tracking transactions on the Ethereum blockchain.

Solidity also supports inheritance, allowing contracts to inherit properties and methods from other contracts. This enables code reuse and modularity, facilitating the development of complex and robust smart contracts.

By understanding the fundamentals of Solidity and its features, developers can create secure and efficient smart contracts on the Ethereum platform. The background knowledge on Solidity provided in this subsection serves as a foundation for the subsequent analysis of smart contract vulnerabilities and the evaluation of ESBMC in this study.

2.2   Smart Contract Vulnerabilities

Smart contract vulnerabilities are security flaws, weaknesses, or design errors in implementing smart contracts, which can lead to unintended behavior or exploitation by malicious actors. Due to the decentralized and transparent nature of blockchain technology and the irreversible nature of transactions on the blockchain, addressing and mitigating these vulnerabilities is paramount for ensuring the security, trustworthiness, and stability of blockchain ecosystems.

Several high-profile incidents, such as the DAO hack in 2016 and the Parity wallet multi-signature vulnerability in 2017 [3], have demonstrated the significant financial and reputational risks associated with smart contract vulnerabilities. These incidents have spurred an increased interest in the research and development of tools and techniques for identifying and mitigating vulnerabilities in smart contracts.

As Ethereum is one of the most widely-used platforms for developing and deploying smart contracts, understanding and addressing vulnerabilities in Ethereum-based smart contracts is crucial for the broader blockchain community.

2.2.1 Smart Contract Weakness Classification (SWC) Registry

The Smart Contract Weakness Classification (SWC) Registry [7] is a comprehensive and well-maintained collection of known vulnerabilities and weaknesses in smart contracts. This registry enables developers, security researchers, and auditors to communicate effectively about smart contract security issues and to foster collaboration in addressing them. The MythX team established the SWC Registry to provide a common language and taxonomy for identifying and categorizing smart contract vulnerabilities.

By classifying smart contract weaknesses into distinct categories, the SWC Registry allows for a better understanding and awareness of these vulnerabilities, facilitating more secure coding practices and robust smart contract development. Each vulnerability listed in the SWC Registry is assigned a unique SWC identifier and includes detailed information, such as its description, impact, potential mitigation, and examples.

Some common vulnerability categories in the SWC Registry are:

- Reentrancy (SWC-107)

- Arithmetic Issues (e.g., Integer Overflow and Underflow) (SWC-101)

- Insecure DelegateCall Implementation (SWC-112)

- Authorization through tx.origin (SWC-115)

As part of this project, the SWC Registry is a valuable resource for understanding smart contract vulnerabilities, their consequences, and mitigation techniques. By referring to the SWC Registry, the benchmark suite of vulnerable smart contracts is developed to cover a wide range of security weaknesses, ensuring a comprehensive evaluation of ESBMC's effectiveness in detecting and mitigating these vulnerabilities.

Moreover, using the SWC Registry's taxonomy and classification system, this study aims to present a structured and systematic analysis of ESBMC's performance, making it easier for other researchers and developers to compare and contrast the results with those of alternative model-checking tools or verification techniques. This, in turn, will contribute to the overall improvement and advancement of smart contract security research and the development of more secure and reliable blockchain ecosystems.

2.3 ESBMC: Efficient SMT-based Bounded Model Checker

The Efficient SMT-based Bounded Model Checker (ESBMC) is a state-of-the-art software verification tool designed to check the correctness and reliability of programs written in C, C++, and Solidity languages. ESBMC employs Satisfiability Modulo Theories (SMT) solvers to perform bounded model checking, a formal verification technique that explores the state space of a program to detect and analyze potential errors, bugs, or security vulnerabilities within a predefined number of execution steps.

2.3.1 ESBMC's Capabilities

ESBMC is equipped with several advanced features and capabilities that make it a powerful and versatile tool for software verification, particularly in the context of smart contract security. Some of the key capabilities of ESBMC include:

- **Bounded Model Checking**: ESBMC uses bounded model checking to explore the state space of a program within a specified depth limit. This technique allows for efficient and scalable verification of complex programs, making ESBMC suitable for analyzing real-world smart contracts with complex logic and interactions.

- **SMT Solvers**: ESBMC employs various SMT solvers, such as Z3, CVC4, and Collector, facilitating efficient decision-making and constraint-solving during the verification process. The integration of these solvers enables ESBMC to effectively analyze complex mathematical operations, data structures, and program behaviors.

- **Assertion Checking and Invariant Generation**: ESBMC can automatically check user-defined assertions and generate invariants for loops and other program constructs. This capability enables the detection of potential errors or vulnerabilities that could violate a smart contract's intended behavior or safety properties.

- **Concurrency Support**: ESBMC provides support for verifying concurrent programs, allowing for the detection and analysis of potential race conditions, deadlocks, and other concurrency-related issues in smart contracts.

With these capabilities, ESBMC has the potential to become an effective tool for detecting and mitigating vulnerabilities in Ethereum-based smart contracts. In this

study, we aim to evaluate ESBMC's performance and effectiveness in identifying and addressing vulnerabilities within a benchmark suite of intentionally vulnerable smart contracts, ultimately contributing to developing more secure and reliable verification techniques for smart contract security.

2.3.2 ESBMC and Solidity

The current implementation of ESBMC's solidity frontend is a developing area, defined as an "early prototype" in the ESBMC documentation [4]. ESBMC does not support the full Solidity language at the time of writing. Currently supported is the ability to check individual functions for many common vulnerabilities. However, reviewing whole smart contracts is not fully supported, nor are many key elements of the Object Oriented Programming (OOP) paradigm in Solidity. These unfinished features will become limiting for many of the vulnerabilities in the SWC Registry [7], as many of them are related to the OOP paradigm or the functionality of a smart contract.

2.3.3 ESBMC's Verification Process

ESBMC takes an input of a solidity smart contract. It then performs a series of steps to verify the smart contract. These steps are shown in Figure 2.1. The first step is to parse the smart contract into an intermediate representation (IR); in this case, we use an AST (Abstract Syntax Tree). The AST is then converted into a GOTO program, processed by the symbolic execution engine (SymEx), giving us a static single assignment (SSA) form. The SSA form is then passed to the SMT Solvers (for example, Bitwuzla, Boolector, CVC4, or Yices), which check for errors. The SMT solver will return a counterexample if it finds an error, which is then used to generate a test case. The test case is then used to generate a trace, which is then used to generate a counterexample. The counterexample is then used to generate a bug report, which is then outputted to the user [9]. Further details on SMT-based solving can be found in *SMT-based bounded model checking for embedded ANSI-C software* [2], and it is beyond the scope of this project.
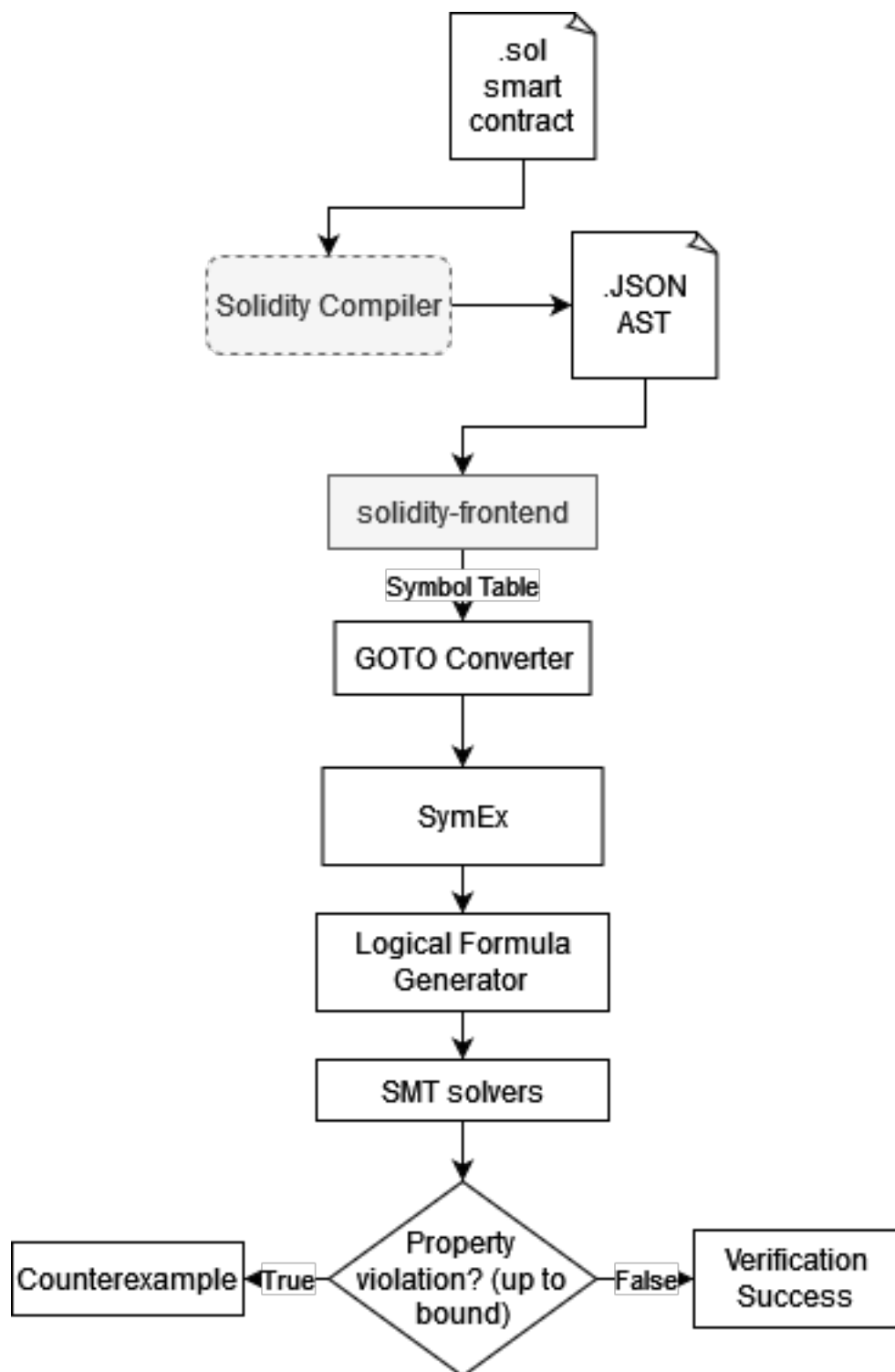
Figure 2.1: ESBMC's Verification Process [9]

Chapter 3

Methodology and Implementation

In this chapter, we outline the methodology and implementation of the project, focusing on the development of a benchmark suite of vulnerable smart contracts that cover a wide range of vulnerabilities listed in the Smart Contract Weakness Classification (SWC) Registry [7]. The purpose of this benchmark suite is to evaluate the capabilities and effectiveness of the ESBMC in detecting and mitigating smart contract vulnerabilities, ultimately contributing to the improvement of verification techniques and tools for smart contract security. By following a systematic and structured approach to create and analyze these vulnerable smart contracts, we aim to provide valuable insights into the performance of ESBMC and identify potential areas for enhancement, fostering the development of more secure and reliable blockchain ecosystems. The benchmark suite does not cover all of the SWC vulnerabilities, but we will discuss the remainder elsewhere, where a more analytical approach is required.

3.1 SWC Registry Vulnerabilities

In the following sections, we will look at each SWC vulnerability, discuss how to implement it in a smart contract, and evaluate how ESBMC tests it. The vulnerabilities have been listed in the order they appear in the SWC registry.

3.1.1 Function Default Visibility

The Function Default Visibility vulnerability (SWC-100) occurs when a function is not explicitly defined as public, internal, external, or private. The default function visibility, in this case, will be set to public which can be a security risk. For example,

```
function defaultVisibility() {
    // This function is public by default.
}
```

Figure 3.1: Extract from a vulnerable contract with Function Default Visibility vulnerability
Source: esbmc-sbms/src/SWC-100/function_default_visibility.sol

```
Error: No visibility specified. Did you intend to add "public"?
```

Figure 3.2: Error message from solc compiler when not setting function visibility

a function that is not intended to be called by other contracts can be called by other contracts if it is not explicitly defined as private.

An example of this vulnerability can be seen in fig. 3.1

Using recent solidity compilers (e.g. solc 0.8.0) the compiler itself will now output an error, as shown in fig. 3.2, if a function is not explicitly defined as public, internal, external, or private. However, this is not the case for older versions of the compiler. This vulnerability would also be solved if a later vulnerability, SWC-102 (discussed in 3.1.3), is implemented. Since the vulnerability occurs when the contract is compiled using an older version of the compiler, when we check the AST of the contract, the function's default visibility will have already been set to public. Therefore, ESBMC will not be able to detect this vulnerability as it may not be able to distinguish between a function that is deliberately public or not.

3.1.2   Integer Overflow and Underflow

The Integer Overflow and Underflow vulnerability (SWC-101) occurs when an arithmetic operation results in a value greater than the maximum value for its type or less than the minimum value for its type. This error can lead to unexpected behavior and security vulnerabilities. For example, an integer overflow can bypass access vulnerable control checks.

An example of this vulnerability can be seen in fig. 3.3

ESBMC is effectively able to check for this vulnerability in solidity smart contracts.

```
//SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

contract IntegerOverflowUnderflow {
    uint256 public counter = 0;

    function increment() public {
        counter += 1;
    }

    function decrement() public {
        counter -= 1;
    }

    function instanceOfIntegerOverflow() public {
        uint256 max = 2**256 - 1;
        uint256 overflow = max + 1;
    }

    function instanceOfIntegerUnderflow() public {
        uint256 min = 0;
        uint256 underflow = min - 1;
    }
}
```

Figure 3.3: Example from a vulnerable contract with Integer Overflow vulnerability
Source: esbmc-sbms/src/SWC-101/integer_overflow_underflow.sol

```
//SPDX-License-Identifier: GPL-2.0
pragma solidity 0.4.13;
```

Figure 3.4: Example from a vulnerable contract with Outdated Compiler Version vulnerability
Source: esbmc-sbms/src/SWC-102/outdated_compiler_version.sol

3.1.3   Outdated Compiler Version

The Outdated Compiler Version vulnerability (SWC-102) means that a legacy compiler version is used to compile the smart contract. This can lead to security vulnerabilities, especially if there are known vulnerabilities in the compiler version. For example, a compiler version that is vulnerable to tx.orign attacks can lead to a contract being exploited, despite the vulnerability largely being fixed in later versions of the compiler.

An example of this vulnerability can be seen in fig. 3.4

As it stands, ESBMC does not check for this vulnerability. However, it is possible to implement this vulnerability by checking the compiler version used to compile the smart contract as the AST does explicitly state the version of the solidity compiler used. If the compiler version is outdated, then it would be considered to be vulnerable. However this would entail cultivating either a list of outdated compiler versions or a list of compiler versions that are vulnerable to known vulnerabilities. This is inherently a challenge since this list would need to be updated regularly, and it is not clear how to determine which compiler versions are vulnerable to which vulnerabilities.
ESBMC-Solidity is still in early stages of development as it stands and currently only supports checking individual functions, as opposed to the entire contract, as is possible in the clang frontend for example. This is a limitation which will be addressed many times in this report since many vulnerabilities fall outside of functions.

3.1.4   Floating Pragma

The Floating Pragma vulnerability (SWC-103) occurs when the compiler version is not explicitly defined in the smart contract. This vulnerability is inherently similar to the Outdated Compiler Version vulnerability (SWC-102) as it can lead to security vulnerabilities, especially if there are known vulnerabilities in the compiler version. For example, a specific compiler version may have a known vulnerability that is fixed in a later version of the compiler or does not yet exist in an earlier version. If this specific compiler version is used to compile the smart contract, then the contract may

```
//SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;
```

Figure 3.5: Example from a vulnerable contract with Floating Pragma vulnerability
Source: esbmc-sbms/src/SWC-103/floating_pragma.sol

```
    "id": 1,
    "literals": [
      "solidity",
      "^",
      "0.8",
      ".0"
    ]
```

Figure 3.6: AST of the contract with Floating Pragma vulnerability
Source: esbmc-sbms/src/SWC-103/floating_pragma.sol

be vulnerable to this known vulnerability.

An example of this vulnerability can be seen in fig. 3.5

Like the Outdated Compiler Version vulnerability (SWC-102), ESBMC does not have a feasible means to check for this issue, given that ESBMC-Solidity currently only supports checking individual functions. Once checking entire smart contracts is implemented, however (as is the case in the clang frontend), this vulnerability can be checked for by checking the compiler version used to compile the smart contract. If the version contains an element of the form ^, then it would be considered vulnerable, as seen in fig. 3.6.

3.1.5 Unchecked Call Return Value

The Unchecked Call Return Value vulnerability (SWC-104) occurs when the return value of a call is not checked. Execution therefore continues as if the call was successful, even if it was not. If the call was not successful, then the execution may be vulnerable to an attack, by breaking the contracts logic.

An example of this vulnerability can be seen in fig. 3.7

TODO: Make a remark here

```
//SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

contract UncheckedReturnValue {
    function callchecked(address callee) public {
        require(callee.call());
    }

    function callunchecked(address callee) public {
        callee.call();
    }
}
```

Figure 3.7: Example from a vulnerable contract with Unchecked Call Return Value vulnerability
Source: esbmc-sbms/src/SWC-104/unchecked_return_value.sol

3.1.6 Unprotected Ether Withdrawal

This vulnerability (SWC-105) can occur due to insufficient access controls. One common occurance of this vulnerability is when a contract has a constructor which is not correctly named, which means that the contract can be reinitialized, perhaps with a different owner which may have differing permissions.

Currently ESBMC-Solidity's implementation of Object-Oriented checking is very limited in nature and does not support checking for this vulnerability. However, it is possible to implement this vulnerability by checking the constructor of the contract. If the constructor is not correctly named, then it would be considered vulnerable. This may involve only accepting the constructor name "constructor". However this is only one part of this vulnerability and it is not clear how to check for the other parts of this vulnerability.

The practical means of fixing a contract with a SWC-105 vulnerability is to ensure that withdrawals and other important functions can only be triggered successfully by some authorized parties. For ESBMC to check for this vulnerability, it would need to be able to check for the existence of access controls, which is not currently possible. ESBMC would also need to be able to detect when a Withdrawal is being made which is much more feasible.

```
//SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;


contract UnprotectedSelfDestructInstruction {
    function destroyAnyone() public {
        selfdestruct(msg.sender);
    }
}
```

Figure 3.8: Example from a vulnerable contract with Unprotected Self-Destruct vulnerability
Source: esbmc-sbms/src/SWC-106/unprotectedSelfDestruct.sol

3.1.7 Unprotected Self-Destruct

The Unprotected Self-Destruct vulnerability (SWC-106) occurs when a contract is able to be self-destructed without any sufficient restrictions. This can lead to the contract being exploited, as the contract can be destroyed by an attacker.

An example of this vulnerability can be seen in fig. 3.8

Currently, ESBMC does not support checking for this vulnerability at all in solidity since the ESBMC-Solidity frontend does not support checking for self-destruct instructions. An implementation may involve checking if a self-destruct instruction is found. If so, it would be considered vulnerable if the function it is held within is public. However, this is not an appropriate implementation since it is possible to have a self-destruct instruction in a private function, which is callable by a public function, which would be vulnerable and would not be detected by this implementation. The implementation would also incorrectly identify a contract as vulnerable if it has a self-destruct instruction in a public function that has sufficient access controls to the actual selfdestruct keyword. Nevertheless, this is a concerning vulnerability and should be addressed in the future.

3.1.8 Reentrancy

Reentrancy (SWC-107) is a vulnerability which is easy to overlook while writing a smart contract. It occurs in Solidity smart contracts when a contract's function, while still in execution, is indirectly called again by an external contract before the initial function call is completed. This vulnerability can lead to unintended behavior and can potentially be exploited by attackers to manipulate the state of the contract or drain its funds.

The root cause of reentrancy vulnerability often lies in the inappropriate ordering of statements in a function. For instance, if a contract's function first transfers funds to an external address (using a low-level function like call) and then updates the internal state (such as the balance of the sender), an attacker might create a malicious contract that, when called by the vulnerable contract, recursively calls the original function before the state update is executed. This allows the attacker to repeatedly withdraw funds without the vulnerable contract realizing the depletion of its balance.

An example of this vulnerability can be seen in fig. 3.9. This withdraw function is vulnerable to reentrancy because the balance of the sender is updated before the transfer is made. This means that if the transfer fails, the balance of the sender will still be updated. If the transfer is successful, the balance of the sender will be updated twice. This is because the transfer function will call the fallback function of the malicious contract, which will call the withdraw function again. This will result in the balance of the sender being updated twice, which is incorrect. Various techniques such as using a mutex locked varable or the "Checks-Effects-Interactions" pattern, can mitigate this vulnerability.

ESBMC-Solidity can already check for reentrancy vulnerabilities [8]. SWC-107, in that case, is implemented successfully.

3.1.9   State Variable Default Visibility

The State Variable Default Visibility vulnerability (SWC-108) occurs when a state variable is not explicitly declared as public, private, or internal. This vulnerability means that the state variable is public by default, which can lead to unintended behavior. An attacker can exploit this vulnerability to manipulate the state of the contract or drain its funds. This vulnerability is fundamentally similar in nature to the Function Default Visibility vulnerability (SWC-103), which is discussed in section **??**.

An example of the vulnerability is seen in 3.10 where the state variable d is internal by default. This may cause unintended behavior if another contract extends the contract. If the contract is extended by another contract, the state variable d will be accessible by the extended contract. This may lead to unintended behavior and can potentially be exploited by attackers to manipulate the state of the contract or drain its funds.

Unlike in the case of the Function Default Visibility vulnerability, the solc compiler does not output an error when a state variable is not explicitly declared as public, private, or internal; instead it is internal by default, and there is no distinguishing

```solidity
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;


contract Reentrancy {

    mapping (address => uint) public balances;

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

    function withdraw(uint _amount) public {
        if (balances[msg.sender] >= _amount) {
            (bool result, ) = msg.sender.call{value: _amount}("");
            require(result);
            balances[msg.sender] -= _amount;
        }
    }


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

Figure 3.9: Example from a vulnerable contract with Reentrancy vulnerability
Source: esbmc-sbms/src/SWC-107/reentrancy.sol

```solidity
contract StateVariableDefaultVisibility {
    uint public a;
    uint internal b;
    uint private c;
    uint d;

    function f() public {
        a = 1;
        b = 2;
        c = 3;
        d = 4;
    }
}
```

Figure 3.10: Example from a vulnerable contract with State Variable Default Visibility
vulnerability
Source: esbmc-sbms/src/SWC-108/state_variable_default_visibility.sol

```
    function foo() public pure {
        assert(false);
    }
```

```
Counterexample:

State 1 file assert_violation.sol line 17 function foo thread 0
----------------------------------------------------
Violated property:
  file assert_violation.sol line 17 function foo
  assertion
  0
```

Figure 3.11: Example from a vulnerable contract with Assert Violation vulnerability
Source: esbmc-sbms/src/SWC-110/assert_violation.sol

marker in the AST to indicate that the state variable is internal by default or by purpose. This means that ESBMC-Solidity cannot check for this vulnerability at the moment.

3.1.10   Unitialized Storage Pointer

Uninitialized Storage Pointer (SWC-109) is a vulnerability that occurs when a storage pointer is not initialized. Therefore uninitialized local storage pointers can point to unexpected locations, leading to unintended behavior. This vulnerability has been resolved in Solidity 0.5.0, thus this vulnerability will be addressed sufficiently if the outdated compiler version vulnerability (SWC-102) discussed in section 3.1.3 is also addressed.

3.1.11   Assert Violation

Well-implemented code should not contain any assert violations. There are two reasons an assertion failure can occur. Either a bug in the contract allows it to enter an invalid state, or the assert statement is used incorrectly. The Assert Violation vulnerability (SWC-110) occurs when an assert statement is used incorrectly.

This vulnerability is shown in 3.11. The assert statement is used incorrectly in this example. The assert statement is used to check if false is true, which is always false. This means that the assert statement will always fail, and the contract will always revert.

ESBMC-Solidity is capable of checking for this vulnerability, as seen in 3.11, providing a counterexample that shows the assert statement is violated. Thus this vulnerability is implemented successfully.

3.1.12 Use of Deprecated Solidity Functions

For a language to maintain backward compatibility with older versions of itself, it is necessary not to change the functionality of existing functions as they are made obsolete or deemed insecure. This way, people with experience using the older language versions can be aware of the changes and update their code accordingly. However, this can lead to the creation of new functions similar to the old ones but with different functionality. This can lead to confusion and cause developers to use the old functions, leading to unintended and insecure behavior. This vulnerability is known as the Use of Deprecated Solidity Functions vulnerability (SWC-111).

An example of some use of deprecated solidity functions can be found in fig. 3.12.

Fortunately, the use of these deprecated functions will cause recent versions of the solidity compiler to throw an error. Therefore this is not a vulnerability ESBMC-Solidity would need to check. However, it would still be possible for a contract that is compiled with an outdated compiler (as discussed in section 3.1.3) to still use these now insecure functions. Therefore, once again, it emphasizes implementing (SWC-102).

3.1.13 Delegatecall to Untrusted Callee

This vulnerability (SWC-112) occurs when a contract calls a function using delegatecall to an untrusted contract. The method `delegatecall` is a variant of a message call, which only differs where the code at the target address is exectued within the contract it is called from (the caller) without changing the values of `msg.sender` and `msg.value`. This should be used sparingly as it allows smart contracts to dynamically load code from untrusted addresses after compilation, and many elements, for example the balance of the contract, can be manipulated by the callee.

ESBMC-Solidity's capacity to check for this vulnerability is limited, since the vulnerability is more of a bad practice in nature and there are many instances where it is acceptable to use delegatecall. For example, it is acceptable to use delegatecall when calling a library function. However, it is not acceptable to use delegatecall when calling a function from an untrusted contract. Therefore, there is no feasible way to check

```
function suicideContract() public {
    // Use selfdesturct(address) instead
    suicide(msg.sender);
}

function callcode() public {
    // Use call() instead
    msg.sender.callcode();
}

function getBlockHash(uint x) public returns(bytes32) {
    return block.blockhash(x);
}

function hash(uint x) public returns(bytes32) {
    return sha3(x);
}

function getGas() public returns(uint){
    return msg.gas;
}

function useThrow() public {
    throw;
}

function set_variables() public {
    constant c = 1;
    var v = 2;
}
```

Figure 3.12: Examples of deprecated solidity functions

```
function failLoop() public {
    for (uint i = 0; i < _addresses.length; i++) {
        require(_addresses[i].send(_balances[_addresses[i]]));
    }
}
```

Figure 3.13: Example from a vulnerable contract with DoS with Failed Call vulnerability

Source: esbmc-sbms/src/SWC-113/DoS_with_failed_call.sol

for this vulnerability in ESBMC-Solidity, as it would involve analyzing whether or not a callee is trusted, which is not feasible.

3.1.14 DoS with Failed Call

External calls will fail sometimes, for example when the called contract runs out of gas. This can be exploited to cause a denial of service. This vulnerability is known as the DoS with Failed Call vulnerability (SWC-113). For example if a contract calls a function in another contract, and the function in the other contract runs out of gas, the contract may revert. This means that the contract will not be able to continue executing, and this may mean that neccesary tasks are not completed.

In figure 3.13, the function failLoop will terminate if any of the calls to send fail. Thus ending the loop prematurely and in this case, not sending all the funds to the addresses.

TODO: AFDSAG

3.1.15 Authorization through tx.origin

Similarly to the Use Of Deprecated Solidity Functions vulnerability (section. 3.1.12 SWC-111), the use of tx.origin for authorization is also deprecated. The tx.origin variable is the address of the account that sent the transaction. If an authorized account calls into a malicious contract that calls back into the main contract using tx.origin, the malicious contract will be able to call functions that it is not authorized to call. This vulnerability is known as the Authorization through tx.origin vulnerability (SWC-115).

An example of this vulnerability can be found in figure 3.14. The function sendTo will only allow the owner of the contract to send ether to another address. However, if the owner calls a malicious contract that calls back into the main contract using tx.origin, the malicious contract will be able to call the function sendTo and send

```
   function sendTo(address payable _to) public {
       require(tx.origin == owner);
       _to.transfer(1 ether);
   }
```

```
Checking function sendTo ...
   - Pattern-based checking: SWC-115 Authorization through
      tx.origin
  checking function body stmt 0
 ERROR: Found vulnerability SWC-115 Authorization through
     tx.origin
 ERROR: VERIFICATION FAILED
```

Figure 3.14: Example from a vulnerable contract with Authorization through tx.origin vulnerability
Source: esbmc-sbms/src/SWC-115/authorization_through_tx_origin.sol

ether to any address.

This vulnerability is very common and a simple fix for ESBMC-Solidity which has already been implemented, as it is a simple pattern-based check [8].

3.1.16   Block Values as a Proxy for Time

The Solidity vulnerability "Block Values as a Proxy for Time" arises when developers rely on block properties, such as block numbers or block timestamps, to measure time durations or control the execution flow of a smart contract. This practice can lead to potential security risks and unintended behaviors due to the inherent variability and manipulability of block-related properties.

Block numbers and timestamps can be influenced by miners to a certain extent, allowing for potential exploitation. For example, a miner might manipulate the block timestamp to impact the outcome of a time-based condition within a smart contract. Using block values as a proxy for time can also result in imprecision due to the variable nature of block times, which might deviate from the expected average block time on the network.

To mitigate this vulnerability, developers should avoid using block properties to control critical time-sensitive operations in smart contracts. Instead, they should consider using alternative mechanisms that are less susceptible to manipulation, such as external time oracles or predefined intervals that do not depend on block values. By ensuring that smart contracts handle time-related operations securely, developers can

```
function lockEth(uint _time, uint _amount) public payable {
        require(msg.value == _amount, 'must send exact amount');
        users[msg.sender].unlockBlock = block.number + (_time / 14);
        users[msg.sender].amount = _amount;
}
```

Figure 3.15: Example from a vulnerable contract with Block Values as a Proxy for Time vulnerability

Source: `https://swcregistry.io/docs/SWC-116#time_locksol` [11]

prevent potential exploitation and maintain the reliability and trustworthiness of the contract.

ESBMC-Solidity can check for this vulnerability by checking if the block number or timestamp is used in a conditional statement. If it is, then the vulnerability is present. This is not yet implemented in ESBMC-Solidity. This is not exhaustive, as there are many other ways to use block values as a proxy for time, but it is a good start. For example in 3.15 the function lockEth uses the block number to determine when the user can withdraw their funds. The line `...unlockBlock = block.number + (_time / 14)` sets the unlockBlock to the current block number plus the time in seconds divided by 14, but future logic involving the unlockBlock variable is not checked. This means that an attacker could use the unlockBlock variable in a conditional statement that ESBMC-Solidity does not check even under this implementation.

3.1.17 Incorrect Constructor Name

Over the period solidity has been developed, there have been many ways to define a constructor for an object. The most common way is to define a function with the same name as the contract, but this is not the only way. In solidity, a constructor can be defined by using the `constructor` keyword, or by defining a function with the same name as the contract. The `constructor` keyword was introduced in Solidity 0.4.22, and the function with the same name as the contract was deprecated in Solidity 0.4.22. It can therefore be a vulnerability to incorrectly define a constructor (SWC-118).

As seen in figure 3.16, the more recent versions of the solidity compiler will output an error if a function with the same name as the contract is defined. However, the older versions of the solidity compiler will not output an error, and will instead define the function with the same name as the contract as the constructor. This means that the function with the same name as the contract will be called when the contract

```
Error: Functions are not allowed to have the same name as the
contract. If you intend this to be a constructor, use
"constructor(...) { ... }" to define it.
```

Figure 3.16: Error outputted when using a function with the same name as the contract

is deployed, and not when the contract is instantiated. This can lead to unexpected behavior, and is a security vulnerability. Once again, this is another issue that can be solved by solving the Outdated Compiler version vulnerability (section **??** SWC-101) or the Floating Pragma vulnerability (section 3.1.4 SWC-102).

This does not however solve all the possible ways of reaching this vulnerability. For example, if a developer is used to constructors' implementation previous to Solidity 0.4.22, they might define a function with the same name as the contract and additionally make a spelling error leading to the creation of a public function which has some capabilities of a constructor (such as being able to receive ether). This is not a vulnerability, but it is a common mistake that developers make. Handling this case is not trivial but is a lower priority issue than other vulnerabilities.

3.1.18   Shadowing State Variables

When Inheritance is used in Solidity, the child contract inherits all the state variables and functions from the parent contract. However, if the child contract defines a state variable with the same name as a state variable in the parent contract, the child contract's state variable will shadow the parent contract's state variable. There would then be a different state variable with the same name in the child contract and the parent contract (SWC-119).

Since inheritance is not a feature that ESBMC-Solidity is capable of checking, given that the object oriented features of Solidity are not yet fully supported, this vulnerability is not yet implemented in ESBMC-Solidity. However, this is a vulnerability that is very common and should be implemented in the future. However the more recent solidity compilers will output a warning if a state variable is shadowed, so this vulnerability can be mitigated by using a more recent version of the solidity compiler (section **??** SWC-101).

3.1.19 Weak Sources of Randomness from Chain Attributes

There are many reasons that it is feasible that a developer may choose to involve randomness in a smart contract (SWC-120). For example, the contract may have a lottery, a game, or a need for random numbers for security. Therefore it is important that the randomness is secure and cannot be manipulated by an attacker. It is difficult to create a strong source of randomness in Ethereum as a large amount of the randomness is determined by the miners. However, there are some ways to create a strong source of randomness, by using a trusted randomness oracle [1] or a commitment scheme. Ethereum does however contain some weak sources of randomness that can be used in smart contracts. These weak sources of randomness are the block hash, the block number, and the block timestamp. These are all weak sources of randomness because they can be manipulated by miners, and thus should not be used in smart contracts as a source of randomness. Solidity does contain a `prevrandao` which replaces the `difficulty` field in the block header.

This is not a vulnerability that ESBMC-Solidity could feasibly check for, as it would require the ability to understand the intention of a program and whether or not the program is using the `prevrandao`, `timestamp` field as a source of randomness, or as other intended uses.

3.1.20 Missing Protection against Signature Replay Attacks

Signature replay attacks are a type of attack that can be used to steal funds from a smart contract. In a signature replay attack, an attacker can use a signature that was intended for a smart contract to only be used a single time, multiple times. For example, suppose they choose to make a particular transaction, say a withdrawal, and receive a signature for verification. In that case, they can use that signature multiple times to withdraw the funds from the smart contract. A lack of this kind of protection is a vulnerability (SWC-121). To avoid this vulnerability, you could, for example, store every message hash used to withdraw funds from the smart contract and check that the message hash has not been used before.

TODO: Discuss ESBMC-Solidity

3.1.21 Lack of Proper Signature Verification

The Lack of Proper Signature Verification vulnerability (SWC-122) occurs when a smart contract does not correctly validate digital signatures, enabling attackers to forge

or manipulate signatures and gain unauthorized access or perform actions on behalf of other users. This vulnerability can lead to severe security breaches and allow malicious actors to compromise the integrity of a smart contract.

Digital signatures are commonly used in smart contracts to authenticate users and prove that they have authorized specific actions or transactions. The Ethereum platform provides a built-in mechanism, the Ethereum-Signed Message (EIP-712), which allows users to sign messages off-chain. This technique reduces the on-chain computational load and gas costs associated with verifying signatures. The built-in `ecrecover` function provided by Solidity can verify the signer's address from a given message hash and its corresponding signature, and using alternative signature verification schemes is not recommended.

Figure 3.17 shows an example of good practice for signature verification. This contract implements a function that verifies the signature of a message hash. The function first splits the signature into its three components: the recovery ID, the R component, and the S component. It then uses the `ecrecover` function to recover the signer's address from the message hash and the signature. The function returns the recovered address.

ESBMC-Solidity currently does not check for SWC-122. Some pattern-based checks could be implemented to detect this vulnerability, but it's unclear how it would be possible to make an exhaustive check for this vulnerability.

3.2

```
pragma solidity ^0.8.0;

contract SignatureVerification {
    function verifySignature(bytes32 messageHash,
    bytes memory signature) public pure returns (address) {
        uint8 v;
        bytes32 r;
        bytes32 s;

        (v, r, s) = splitSignature(signature);
        return ecrecover(messageHash, v, r, s);
    }

    function splitSignature(bytes memory sig)
    internal pure returns (uint8 v, bytes32 r, bytes32 s) {
        require(sig.length == 65, "Invalid signature length");

        assembly {
            r := mload(add(sig, 0x20))
            s := mload(add(sig, 0x40))
            v := byte(0, mload(add(sig, 0x60)))
        }
    }
}
```

Figure 3.17: Example of a contract that verifies signatures

Bibliography

[1] P. Allemann, C. Cachin, L. Zanolini, and I. A. Sesar. Randomness and games on ethereum. 2021.

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

[3] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IW-BOSE)*, pages 19–25. IEEE, 2018.

[4] ESBMC. ESBMC documentation. Retrieved April 19, 2023, from `http://esbmc.org/`, 2020.

[5] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Roşu. Kevm: A complete semantics of the ethereum virtual machine. 2017.

[6] IBM. Blockchain for financial services. `https://www.ibm.com/blockchain/industries/financial-services`.

[7] MythX. Swc registry. `https://swcregistry.io/`, 2020.

[8] B. Salim. ESBMC based ethereum smart contract security verification. Master's thesis, The University of Manchester, 2022.

[9] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro. Esbmc-solidity: an smt-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 65–69, 2022.

[10] M. Xu, X. Chen, and G. Kou. A systematic review of blockchain. *Financial Innovation*, 5(1), 2019.

[11] K. Zipfel. time_lock.sol. Retrieved March 20 2023, from `https://swcregistry.io/docs/SWC-116#time_locksol`".

Appendix A

AppendixExample