AN ANALYSIS OF ESBMC'S EFFECTIVENESS ON SWC VULNERABILITIES

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

John Lindon Robinson

Department of Computer Science

Contents

Word Count: 2500

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... 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 [6] 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 [8] [5]. 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 [4]. 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 [2], 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 [6] 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 [3]. 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 [6], 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 [7]. Further details on SMT-based solving can be found in *SMT-based bounded model checking for embedded ANSI-C software* [1], and it is beyond the scope of this project.

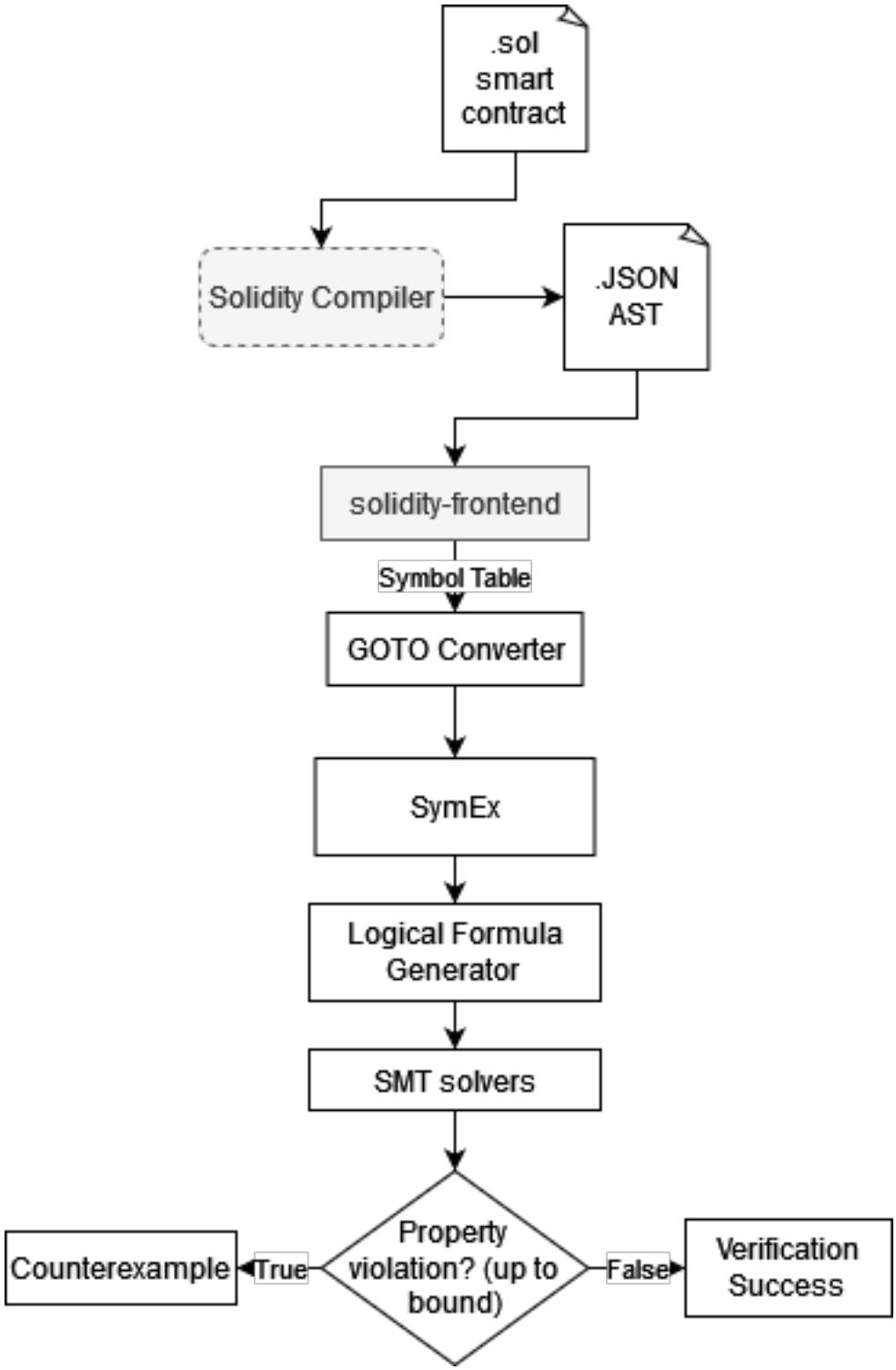

Figure 2.1: ESBMC's Verification Process [7]

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 [6]. 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

3.1.1.1 Vulnerability Description

The Function Default Visibility vulnerability (SWC-100) occurs when a function is not explicitly defined as public, internal, external, or private. The default function

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

contract FunctionDefaultVisibility {
    // Functions that do not specify
    // visibility default to public.
    // This can lead to vulnerabilities
    // if the function is not
    // intended to be public.

    function publicFunction() public {
    }

    function privateFunction() private {
    }

    function internalFunction() internal {
    }

    function externalFunction() external {
    }

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

Figure 3.1: Example of a vulnerable contract with Function Default Visibility vulnerability

visibility, in this case, will be set to public which can be a security risk. For example, 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.

3.1.1.2 Implementation in ESBMC

Bibliography

[1] 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.

[2] 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 (IWBOSE)*, pages 19–25. IEEE, 2018.

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

[4] 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.

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

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

[7] 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.

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

Appendix A

AppendixExample