TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

David, Isaac Mayowa 194441IVSB

An Evaluation Framework for Smart Contract Vulnerability Detection Tools on the Ethereum Blockchain

Bachelor Thesis

Technical Supervisor
Alexander Norta
PhD
Academic Supervisor
Toomas Lepikult
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

David, Isaac Mayowa 194441IVSB

Nutilepingute haavatavuste tuvastamise tööriistade hindamisraamistik ethereum blockchainis

Bakalaureusetöö

Juhendaja
Alexander Norta
PhD
Kaasjuhendaja
Toomas Lepikult
PhD

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author:	David, Isaac Mayowa
Date:	May 16, 2022

Abstract

Blockchain technology has gained traction in a variety of industries due to its potential to revolutionize how data is stored and transactions are processed. Ethereum, the second most popular blockchain platform, has drawn developers to its platform for the purpose of developing smart contracts. These contracts are self-executing programs that reside on the blockchain and are intended to automate specific tasks. The number of smart contracts established over the last few years has been steadily increasing. These contracts are used for a variety of functions, from electronic voting to the storage of medical information and decentralized financial systems. Although the popularity of smart contracts has increased, there has been a corresponding increase in the number of vulnerabilities found in these contracts. Due to the fact that these vulnerabilities have resulted in the loss of millions of dollars in cryptocurrency assets, early vulnerability discovery is critical for mitigating the hazards posed by vulnerable smart contracts.

While numerous tools have been developed to address various types of vulnerabilities, an efficient evaluation framework for determining the utility of smart contract vulnerability detection tools remains lacking. For many smart contract developers, determining which tool or combination of tools is best suited for vulnerability detection remains a challenge.

This thesis addresses the existing gap by proposing an efficient evaluation framework to determine the utility of smart contract vulnerability detection tools. The proposed framework is then evaluated against a set of open-source vulnerability detection tools used in smart contract development. In addition, a state of art review of the most common Ethereum smart contract vulnerabilities is also presented.

This thesis is written in English and is 86 pages long, including 7 chapters, 11 figures and 10 tables.

List of abbreviations and terms

API	Application Programming Interface
CGI	Control Flow Graph
CLI	Command Line Interface
dApps	Decentralized applications
DASP	Decentralized Appliaction Security Project
DeFi	Decentralized Finance
DSR	Design Science Research
DoS	Denial of Service
EVM	Ethereum Virtual Machine
GUI	Graphic User Interface
IOT	Internet Of Things
IS	Information Science
POC	Proof of Concept
UI	User Interface
VM	Virtual Machine

Table of Contents

List of Figures

List of Tables

1. Introduction

Blockchain is a shared distributed ledger that uses a list of ordered entries and records known as *blocks* to keep track of transactions. Each block is interconnected and timestamped, forming a chain. This enables blockchain databases to securely record data transactions without the assistance of a central administrator [1]. Satoshi Nakamoto introduced the concept of the blockchain in 2008, and it was implemented when a new digital currency known as Bitcoin was created using the technology [2]. Blockchain technology is not limited to cryptocurrencies; it has a plethora of applications in a variety of industries. These applications cover a broad spectrum of industries, from banking and finance to healthcare and logistics. Ethereum, the second most popular cryptocurrency, was launched in 2015 by Vitalik Buterin with the goal of enabling smart contracts, since then, the Ethereum blockchain has been used to implement millions of smart contracts. [3]. Smart contracts have a variety of use cases, including multi-signature accounts, data storage, and financial data encoding. Smart contracts are created by encoding rules and instructions in code and then deploying them to the Ethereum blockchain, where they become immutable and enforced by the Ethereum network. Smart contracts have facilitated the development of decentralized applications, or dApps, that are functionally equivalent to traditional software applications, however most smart contracts being computer programs are prone to vulnerabilities, these weaknesses have resulted in the loss of millions of dollars' worth of cryptocurrency holdings.

1.1 Thesis Objectives

Thus, the objective of this thesis is to develop a comprehensive evaluation framework for tools for detecting smart contract vulnerability on the Ethereum blockchain. The following research objectives have been formulated to this end:

1. To conduct a comprehensive review of the most common Ethereum smart contract vulnerabilities
2. To develop an evaluation framework for smart contract vulnerability detection tools
3. To identify and evaluate smart contract vulnerability detection tools that are commonly used on the Ethereum blockchain.

1.2 Existing body of knowledge

The proceeding sections provide a review of existing work on blockchain technology, smart contracts and Ethereum, and then a review of the existing body of knowledge on smart contract vulnerability detection tools.

1.2.1 Introduction to Blockchain Technology

In 2009, the first blockchain was created to serve as the foundation for Bitcoin, the world's first cryptocurrency. Bitcoin is a decentralized digital currency that can be used to purchase and sell both physical and digital goods and services. Bitcoin is a peer-to-peer network, with transactions verified by nodes, or computers that keep a copy of the blockchain. [2]. Blockchain technology has advanced tremendously since its inception in 2009, due to the popularity of Bitcoin and other cryptocurrencies such as Ethereum and Litecoin, blockchain technology has been adopted in a variety of other fields. A traditional database stores data centrally and is administered by a central authority. As a result of this centralization, the database becomes susceptible to data breaches and manipulation. By storing data in a distributed manner, blockchain technology circumvents these vulnerabilities. In a blockchain database, data is stored in blocks, and each block is chained to the previous block, forming a chain. This chain is replicated across a network of computers, with each computer maintaining its own copy. Due to the immutability and distributed nature of blockchain, it is impenetrable to tampering, as any attempt to modify data in a single block results in a discrepancy in the copy of the chain stored on other computers [4]. Blockchain technology enables the development of a diverse range of applications, from digital currencies to supply chain management. [5].

1.2.2 Ethereum

Ethereum is a blockchain that enables the execution of smart contracts, which are decentralized applications that execute exactly as programmed without the need for intermediaries. Vitalik Buterin, a Russian-Canadian programmer, proposed Ethereum in 2013 [6]. Ethereum was funded through a crowdsourcing campaign in 2014 and launched on 30 July 2015 [7]. Ethereum, like Bitcoin, is based on a blockchain, but Ethereum's blockchain is more versatile, allowing for the creation of smart contracts and decentralized applications, or dApps. Ethereum makes use of its own cryptocurrency, Ether, to pay for transaction fees and gas [8].

Ethereum has attracted developers from all over the world who build decentralized applications on the Ethereum blockchain. These decentralized applications have a wide range of applications, from storage and management of medical records to decentralized financial systems [9].

The popularity of Ethereum has resulted in the development of many tools and frameworks to facilitate the development of decentralized applications. These tools and frameworks include Truffle, Embark, and Hardhat.

1.2.3 Smart Contracts

On the blockchain, smart contracts are self-executing contracts. A smart contract is made up of a series of rules and conditions that must be satisfied for the contract to be executed. [10].

Nick Szabo proposed the idea of smart contracts in 1996. Szabo described smart contracts as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [11]. Decentralized applications, or dApps, are frequently built using smart contracts. A decentralized application (dApp) is a program that runs on a decentralized network. The Ethereum Blockchain is a popular choice for many decentralized application developers [8].

The use of smart contracts eliminates the need for a third party to mediate a contract, such as a bank or a lawyer. This not only reduces the contract's cost, but also the time required to complete the contract. Smart contracts are stored on the blockchain and are immutable, which means they cannot be modified once deployed. [10].

1.2.4 Vulnerabilities in smart contracts

Despite their numerous benefits, smart contracts are not without flaws. These flaws can be used by attackers to take control of the contract or steal the funds contained within. Due to the blockchain's immutability, the majority of smart contracts cannot be modified to address vulnerabilities. This demonstrates how critical it is to identify and mitigate vulnerabilities prior to deploying a smart contract.

In June 2016, one of the earliest widely publicised smart contracts on the Ethereum blockchain, known as the DAO, was attacked, approximately $50 million in ETHER was lost as a result. The attacker took advantage of a recursive call bug to gain control of the DAO contract and drain its funds. This hack prompted the Ethereum community's contentious decision to hard-fork the Ethereum blockchain, resulting in the blockchain being split into two branches in order to recover funds lost in the attack [12].

Numerous studies have been conducted in the past attempting to analyze the security risks associated with smart contracts [13][14][15], however the majority of the research work doesn't delve into the technical details of the most prevalent vulnerabilities.

Reentrance attacks, indirect execution of unknown code via a fallback function, interface naming issues, and time component attacks are just a few examples of common smart contract vulnerabilities [13].

1.2.5 Smart Contract Vulnerability Detection Tools

To address the vulnerabilities in smart contracts, several tools for detecting these vulnerabilities have been developed. These tools are classified as static analysis tools or dynamic analysis tools.

Static analysis tools examine a smart contract's source code and do not require the contract to be deployed on a blockchain. Oyente, Mythril, and Securify are all popular examples [16], while contracts that have been deployed on a blockchain are analysed using dynamic analysis tools. An example of a dynamic analysis tool is Manticore. Existing research focuses more on the requirements and methods for developing new vulnerability scanners but very few details are provided on how to access their utility [14], [15], [17]–[19].

1.3 Research Gap

The existing gap lies in the lack of an evaluation method to determine the utility of the numerous smart contract vulnerability tools available for use to smart contract developers.

1.4 Research Methodology and Research Questions

The design science research method was chosen for this study because it is a systematic and iterative method for developing, evaluating, and applying innovative solutions to real-world problems. [20]. In design science research the design process involves a set of expert activities with the end goal being an innovative product. DSR utilizes a knowledge base and a business need around a subject and then iteratively develops theories and artefacts in order to arrive at a viable solution. [20].

1.4.1 Design Science Research Theory

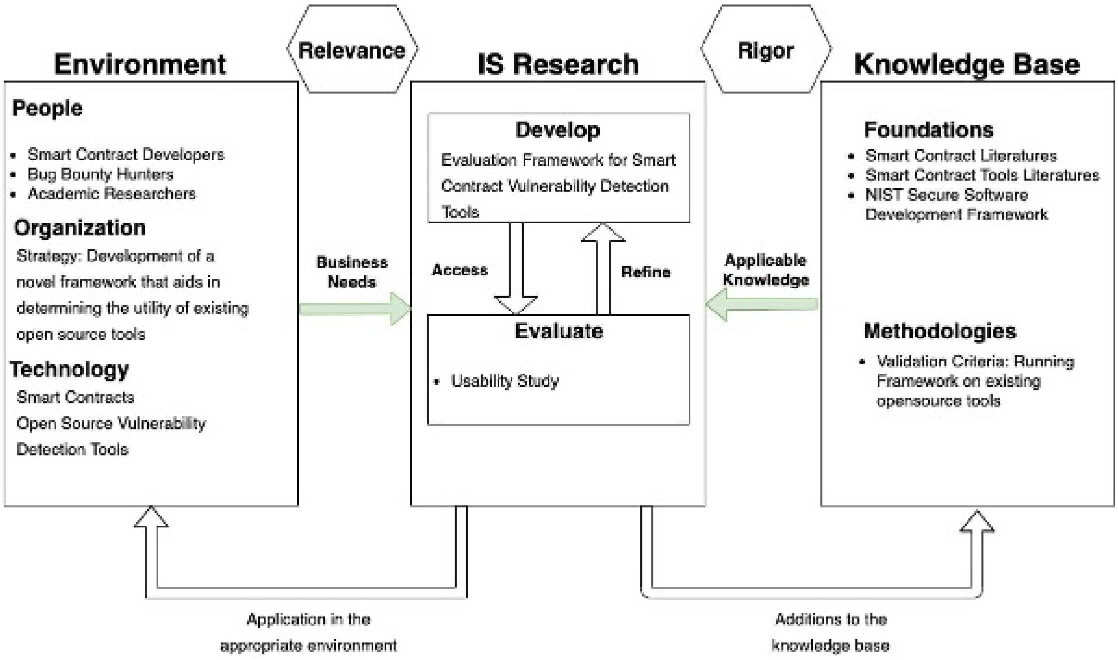

Environment
People
Smart Contract Developers
Bug Bounty Hunters
Academic Researchers
Organization
Strategy: Development of a novel framework that aids in determining the utility of existing open source tools
Technology
Smart Contracts
Open Source Vulnerability Detection Tools
Relevance
Business Needs
IS Research
Develop
Evaluation Framework for Smart Contract Vulnerability Detection Tools
Access
Refine
Evaluate
Usability Study
Rigor
Applicable Knowledge
Knowledge Base
Foundations
Smart Contract Literatures
Smart Contract Tools Literatures
NIST Secure Software Development Framework
Methodologies
Validation Criteria: Running Framework on existing opensource tools
Application in the appropriate environment
Additions to the knowledge base

Figure 1. DSR Theory Overview [20].

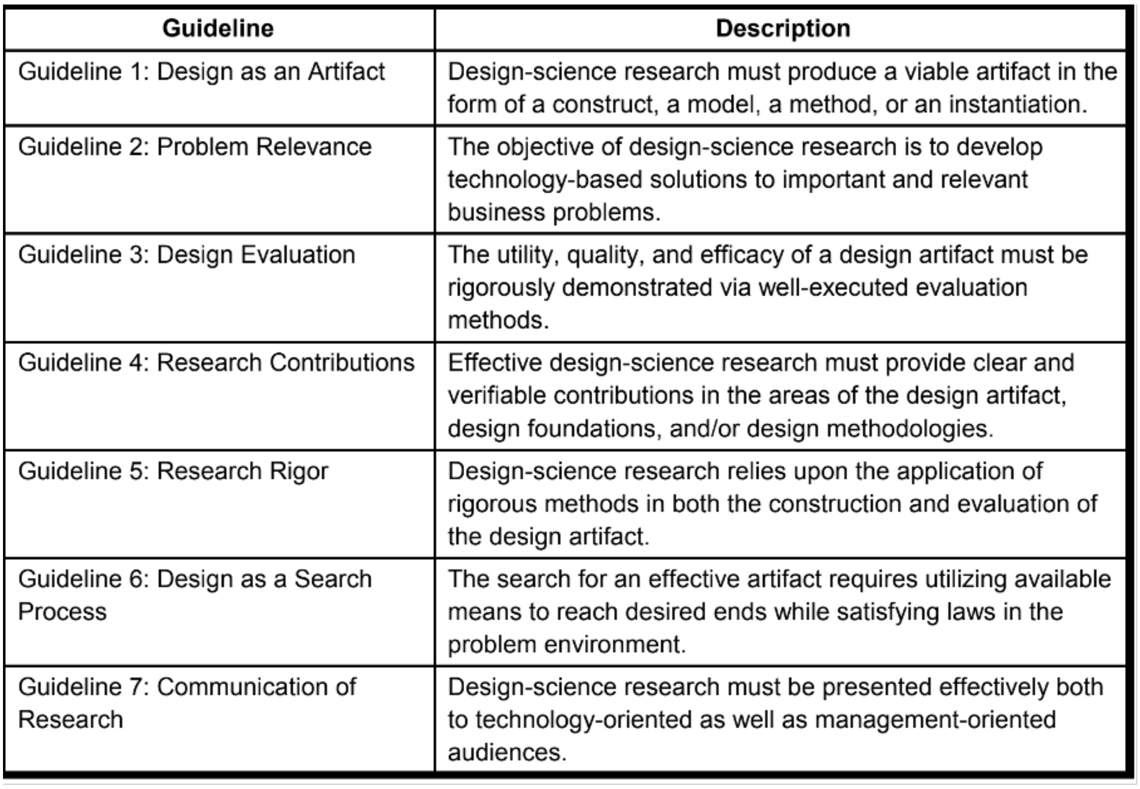
The visualization in Figure 1 depicts the framework for the DSR process for the evaluation framework use case [20], the goal of the design science research is utility, the realm of IS research focuses on the intersection of people, organisations, and technology. [20], the people in this case are smart contract developers, smart contract bug bounty hunters and academic researchers.

The middle column in Figure 1 shows the affiliation to IS research where either an artifact or a theory is produced through iterations of development and evaluation, for this thesis an artifact in the form of an evaluation framework for smart contract vulnerability detection tools is produced. The knowledge base pillar provides the foundation for developing this framework, and the methodology applied is the validation criteria in which the framework developed is used to evaluate the existing opensource tools.

Guidelines for conducting DSR have been provided by Hevner et al [20]. The following sections will show how those guidelines are applicable to the work done in this thesis

1.4.2 Design as an Artifact

The artifact of this DSR project is the evaluation framework for smart contract vulnerability detection tools, The framework consists of a set of metrics and indicators that can be used to evaluate the utility of these tools.



Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 1. Guidelines for conducting DSR [20].

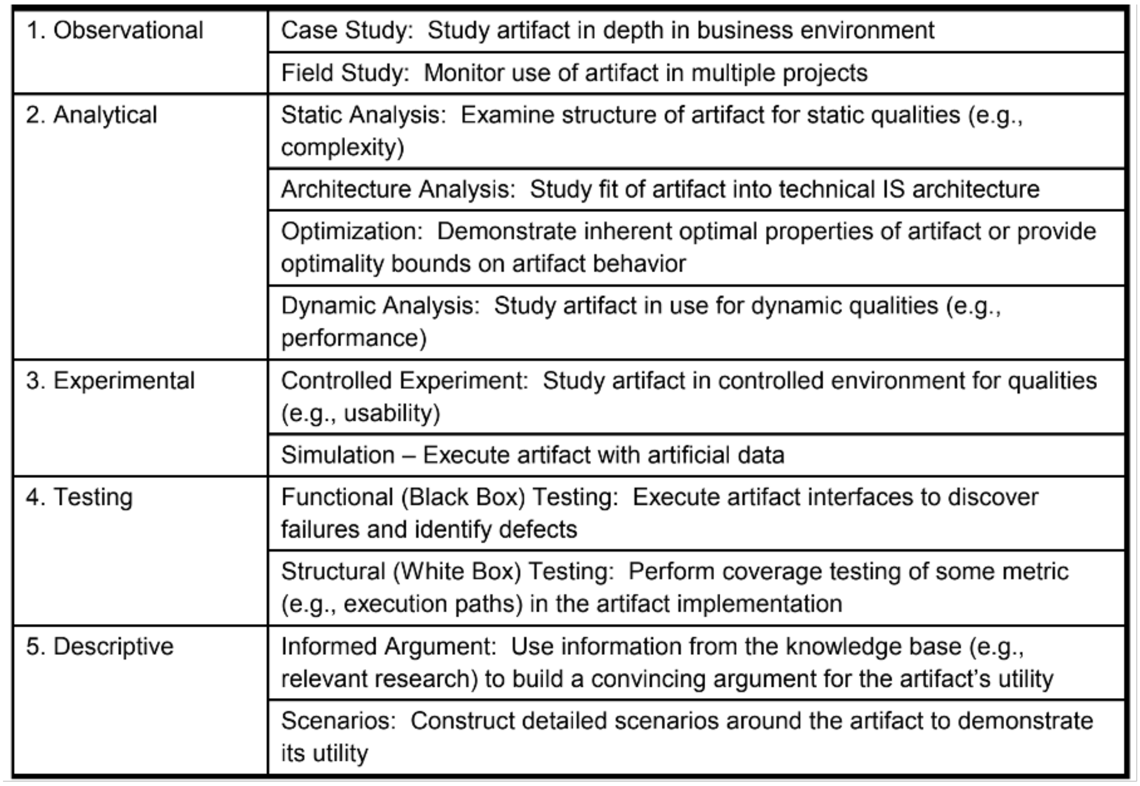
1.4.3 Problem Relevance

Generally, DSR should focus on problems that need to be solved in order to improve the state of practice or knowledge in a specific domain. The problem addressed by this DSR project is how can we effectively evaluate and compare the numerous smart contract vulnerability detection tools available for use by developers.

1.4.4 Design Evaluation

Throughout the DSR process, evaluation should be conducted. Evaluation is a critical component of DSR because it enables researchers to determine the effectiveness of their designs and whether they achieved their objectives [20]. According to Hevner et al *"IT artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes"* [20]. Table 2 represents the various existing methodologies to evaluate the utility of the artifact

The evaluation framework is subjected to a descriptive review by determining the utility of existing open-source smart contract vulnerability detection tools.



1. Observational	Case Study: Study artifact in depth in business environment
	Field Study: Monitor use of artifact in multiple projects
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity)
	Architecture Analysis: Study fit of artifact into technical IS architecture
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior
	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance)
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability)
	Simulation – Execute artifact with artificial data
4. Testing	Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility

Table 2. DSR Evaluation Methods [20].

1.4.5 Research Contribution

This thesis seeks to contribute to the body of knowledge in three significant areas. Firstly, a comprehensive review of the most common Ethereum smart contract vulnerabilities is presented. Secondly, an evaluation framework for smart contract vulnerability detection tools on the Ethereum blockchain is proposed. Finally, a systematic review of existing smart contract vulnerability detection tools is conducted, and the findings are summarized and presented.

1.4.6 Research Rigor

The DSR process should be conducted in a rigorous and systematic manner. In order to ensure the rigor of this research, the following methods were employed:

- A comprehensive review of existing work was conducted in order to gain a better understanding of blockchain technology, smart contracts, Ethereum and smart contract vulnerability detection tools.
- The proposed evaluation framework was evaluated by conducting an experimental review on existing open-source smart contract vulnerability detection tools.

1.4.7 Design as a Search Process

DSR should be seen as a search process in which researchers iteratively develop and evaluate designs in order to find an effective solution to a problem [20]. The proposed evaluation framework was developed iteratively, with the first iteration focusing on the creation of metrics and indicators for evaluating smart contract vulnerability detection tools. The second iteration utilized the proposed framework to conduct an experimental review of existing open-source smart contract vulnerability detection tools.

1.4.8 Communication of Research

The research findings are communicated in the form of a written report and a presentation. The report details the research process, the proposed evaluation framework, and the experimental review findings. The presentation summarizes the research process and results.

1.5 Research Questions

Based on the previous sections the main research question is formulated: **How can Ethereum Smart Contract Vulnerability Detection tools be evaluated?** In order to answer this question, the following research questions were formulated to guide this research:

1. **RQ1: What are the common vulnerabilities affecting smart contracts on the Ethereum blockchain?**
2. **RQ2: What are the evaluation requirements for vulnerability detection tools on the Ethereum blockchain?**
3. **RQ3: What are the features of the existing open-source vulnerability detection tools on the Ethereum blockchain?**

RQ1 is answered by conducting a comprehensive review of existing work on Ethereum smart contract vulnerabilities.

RQ2 is answered by developing an evaluation framework for smart contract vulnerability detection tools that can be used to assess the utility of these tools.

RQ3 is answered by conducting a review of open-source smart contract vulnerability detection tool.

1.6 Thesis Structure

The remainder of the thesis is structure as follows, Chapter 2, sets the preliminaries of the work and also describes the running case, In Chapter 3 an analysis of the common smart contract vulnerabilities on the Ethereum blockchain will be conducted, Chapter 4 develops the framework and a suitable matrix for evaluating smart contract vulnerability tools on the Ethereum blockchain, In Chapter 5, we explore the features of existing smart contract vulnerability detection tools, In Chapter 6 a prototype of the evaluation framework will be developed and lastly Chapter 7 discusses the limitations and future work possibilities.

2. Preposition

The following sections presents an overview of the prepositions and concepts applied to this thesis, Section 2.1 explains the running case, and problem the thesis is trying to solve, Section 2.2 highlights the importance of early vulnerability detection in smart contract development, Section 2.3 describes the framework's use case in-depth

2.1 Running Case

Smart Contract security is a significant impediment to adoption, as evidenced by blockchain's immutability and the fact that no one writes perfect code. Solidity is the most widely used programming language for developing Ethereum-based smart contracts. [21]. Solidity was proposed in 2014 by Gavin Wood and was released to the public in August 2015 [22], while Solidity, a high-level Turing-complete language, has gained popularity due to its ease of use and flexibility for developers, Solidity is error-prone and has been cited as a significant reason for the error-prone implementation of smart contracts due to its lack of centralized vulnerability documentation and blockchain domain specific constructs [23]. Researchers estimate that approximately 45% of Solidity-based smart contracts are vulnerable [24]. It should be noted that vulnerability does not always imply exploitability [25]. Daniel et al. discovered that only 1.98 percent of 23,327 vulnerable contracts were exploited, resulting in a loss of only 2% of the 3 million Ether at stake [25]. This may appear to be good news, but over $400 million was lost in the first quarter of 2022 due to smart contract vulnerabilities, with the majority of losses occurring on Defi platforms.

Decentralized finance, or Defi, is a new category of decentralized applications (DApps) built on Ethereum and other blockchains that provide financial services such as lending, borrowing, derivatives trading, and payments. One of the most popular types of Defi is Decentralized Exchanges (DEXs), which allow for the decentralized trading of cryptocurrencies and other financial assets [26]. Decentralized exchanges have gained popularity among investors and traders because they eliminate the need for a third party to hold or process transactions. Decentralized Exchanges are built on top of existing decentralized protocols and operate using smart contracts to store and process transactions. Due to the rapid growth of Defi as a result of the rapid growth of Ethereum's value and the growing popularity of non-fungible tokens (NFTs), on the Ethereum blockchain, the number of smart contracts implemented has risen dramatically. On Ethereum, over 1.5 million smart contracts have been deployed [27], the majority of which are on Defi platforms. The popularity of Defi can be attributed to several factors, including the fact that it is a trustless system, it is transparent, and it is accessible to anyone with an Internet connection.

The Defi ecosystem was valued at $240 billion in December 2022, up from $18 billion in January 2021, a staggering 1222% jump, in a single year [28]. There are currently over $100 billion in Ethereum smart contract assets [29]. Defi and Ethereum are expected to continue growing in popularity as more people recognize the potential of decentralized finance in providing permissionless financial services. Defi's popularity has attracted a lot of attention from hackers, and as a result, several smart contract vulnerabilities have been exploited.

Defi platforms are usually launched hastily without a proper security audit, running a full audit before launch can cost a lot, and since most Defi projects are early and without a budget the amount needed can be out of reach for early Defi companies, this leaves most defi projects vulnerable to hacks and attacks.

Since the well-publicized DAO hack in 2016, the number of hacks as a result of smart contract vulnerabilities has increased. Because smart contracts typically store cryptocurrencies and financial assets, smart contract exploits typically result in financial losses. From 2020 to November 2021, Elliptic estimates that over $12 billion was lost to Defi hacks as a result of smart contract vulnerabilities [30]. Most of these vulnerabilities could have been detected and prevented earlier in the development cycle. This data sheds light on Defi's growth and sends a message about the critical nature of smart contract security.

2.2 Importance of Early Vulnerability Detection in Smart Contract Development

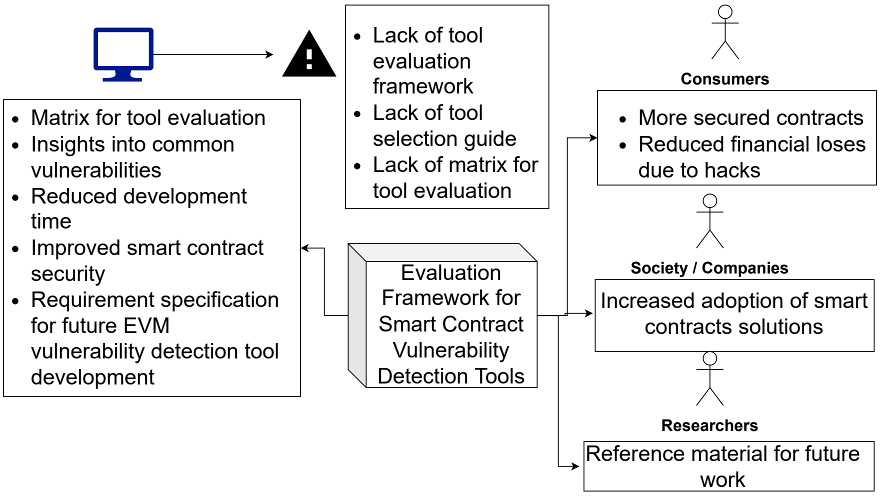
Security is of utmost importance in the blockchain ecosystem, most blockchain developers understand the importance of auditing the security of their contracts before deploying them to the mainnet, due to constraining factors such as the high cost of auditing, they only conduct a full audit at launch. These security audits frequently reveal serious vulnerabilities that could have been avoided if they were identified earlier in the development cycle. The importance of early detection of vulnerabilities in smart contracts has been demonstrated by a number of high-profile hacks. The DAO hack in 2016, which resulted in the loss of $60 million [12], could have been prevented if the vulnerabilities in the code had been identified earlier. The Parity Wallet hack in 2017 [31], which resulted in the loss of $30 million, was also the result of vulnerabilities that could have been identified earlier. These hacks have demonstrated the importance of early detection of vulnerabilities in smart contracts

Early detection of vulnerabilities is essential to the success of any software development project. Early detection is even more critical in the case of smart contracts due to the blockchain's immutability. Once a smart contract is deployed, changing the code is extremely difficult, if not impossible. This means that any vulnerabilities in the code that exist at the time the smart contract is deployed will persist until the smart contract is retired. Numerous tools have been developed in recent years to assist developers with early vulnerability detection. For example, static analysis tools such as Mythril and Oyente can assist developers in identifying a variety of vulnerabilities, including those in the Solidity code itself and the underlying Ethereum Virtual Machine (EVM). Additionally, a few studies have been conducted to examine the feasibility of using machine learning to conduct vulnerability research on smart contracts.

In general, as more work is done and more tools are released to improve smart contract vulnerability detection, a method for determining the utility of the numerous tools available is required.

2.3 The Evaluation Framework

This thesis proposes a methodology for evaluating smart contract security tools. A set of metrics for evaluating the performance of smart contract security tools is also provided.



• Lack of tool evaluation framework
• Lack of tool selection guide
• Lack of matrix for tool evaluation
• Matrix for tool evaluation
• Insights into common vulnerabilities
• Reduced development time
• Improved smart contract security
• Requirement specification for future EVM vulnerability detection tool development
Evaluation Framework for Smart Contract Vulnerability Detection Tools
Consumers
• More secured contracts
• Reduced financial loses due to hacks
Society / Companies
Increased adoption of smart contracts solutions
Researchers
Reference material for future work

Figure 2. Benefits of the framework.

Figure 2 illustrates how developers, businesses, researchers, consumers, and the broader community can benefit from a framework for evaluating the utility of Ethereum smart contract vulnerability detection tools.

On the left column, developers are presented with a Matrix for tool selection when creating a new smart contract on the Ethereum blockchain, this reduces development time because developers can already determine which tool is appropriate for their use case; additionally, the Thesis can serve as a requirement specification for future EVM vulnerability tool development. This can result in a reduction in financial losses due to hacks for smart contract consumers as more secure smart contracts are released. In general, when smart contracts are more secure, confidence is increased, and adoption may increase. Finally, academic researchers can benefit from this thesis's work because it can be used as a reference for future work.

2.4 The Utility Evaluation Web Application

On the basis of the issues outlined above, a web application prototype will be developed. The web application prototype will be based on the ideas developed in this thesis and will allow smart contract developers and general users to easily evaluate the utility of a smart contract vulnerability detection tool. The web application will be developed as a fully functional and usable web application that developers can use to determine the effectiveness of a smart contract vulnerability detection tool based on the evaluation metric developed in this thesis.

Retool and JavaScript will be utilized to create the web application's user interface. The web application will include a tab for weights, a tab for evaluation, and a page for results. The weights tab will be used to assign weights. On the evaluation tab, details of the smart contract vulnerability detection tool to be evaluated will be entered. The evaluation results will be displayed on the results page.

3. Smart Contract Vulnerabilities on the Ethereum Blockchain

This chapter discusses the various vulnerability classification taxonomies on the Ethereum blockchain. Additionally, a holistic analysis of the various types of smart contract vulnerabilities on the Ethereum blockchain will be conducted. Also, each vulnerability will be paired with a plausible exploit scenario including methods for mitigation and remediation.

3.1 Introduction

The objective of this section is to answer Research Question RQ1 - **What are the common vulnerabilities affecting smart contracts on the Ethereum blockchain?**

In Section 3.2, we review prior research on smart contract vulnerabilities, Section 3.3, will discuss the most common vulnerabilities affecting smart contracts on the Ethereum Network using the DASP Top 10 taxonomy, we also provide details on real-world attack scenarios and discuss common remediation techniques.

3.2 Smart Contract Vulnerability Taxonomy

Numerous research has been done on how to group and classify smart contract vulnerabilities, one of the prominent taxonomy is proposed by Atzei et al [32]. In the research done by Atzei et al [32], vulnerabilities are grouped into 3 classes, which are Solidity, EVM bytecode, or blockchain. The classes represent the level in which the vulnerability occurs, In the first class (Solidity), the vulnerabilities reside in the smart contract programming language, In the second class (EVM bytecode), vulnerabilities are in the bytecode produced by the Solidity compiler and are executed on the Ethereum blockchain by the EVM, In the third class(Blockchain) vulnerabilities occur at a higher level and take advantage of the features of the Ethereum. Table 3 summarizes the vulnerabilities and their causes.

Level	Cause of Vulnerability
Solidity	Call to the unknown
	Gasless send
	Exception disorders
	Type casts
	Reentrancy
	Keeping secrets
EVM	Immutable bugs
	Ether lost in trasfer
	Stack size limit
Blockchain	Unpredictable state
	Generating randomness
	Time constraints

Table 3. Taxonomy of Smart Contract Vulnerabilities [32].

The Taxonomy proposed by Atzei et al [32] is a bit outdated as issues such as Stack size limit is now fixed on the EVM.

In the work done by Zulfiqar et al [33], smart contract vulnerabilities are classified into six categories based on the domain knowledge of the operations involved, vulnerabilities are classified into six classes: inter-contractual vulnerabilities, contractual vulnerabilities, arithmetic bugs, gas related issues, transactional vulnerabilities, depreciated vulnerabilities and randomization vulnerabilities.

- **Inter-contractual vulnerabilities** occur as a result of faulty communication between two contracts.
- **contractual vulnerabilities** are bugs that affect the contract itself.
- **arithmetic bugs** refer to vulnerabilities that occur as a result of faulty arithmetic operations.
- **gas related issues** refer to vulnerabilities that result from the misuse of gas.
- **transactional vulnerabilities** refer to vulnerabilities that arise as a result of faulty transaction processing.
- **depreciated vulnerabilities** refer to vulnerabilities that result from the use of deprecated ERC standards and finally, randomization vulnerabilities refer to vulnerabilities that result from the use of random numbers in the contract.

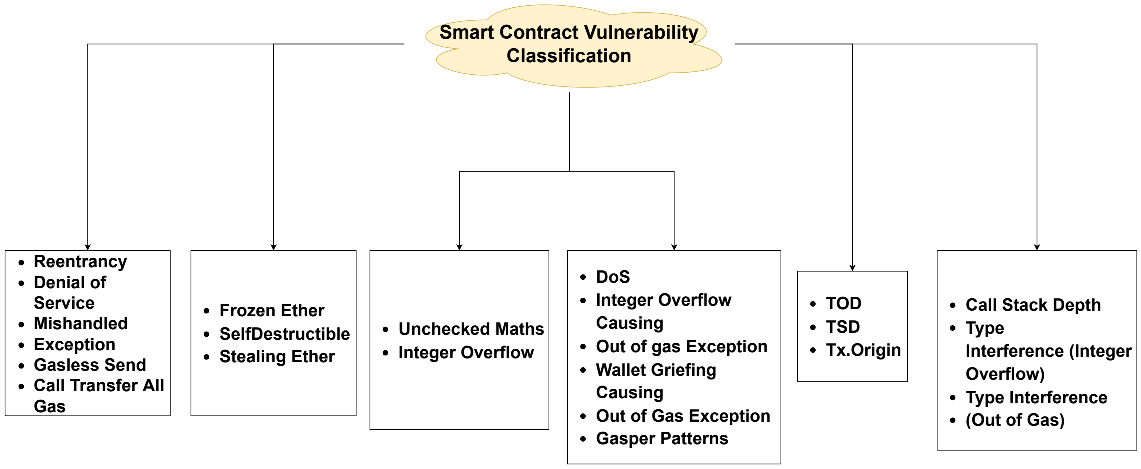


Figure 3. Smart Contract Vulnerability Classification by Zulfiqar et al *[33].*

The classification is summarized in Figure 3. While Zulfiqar et al [33] taxonomy is more comprehensive and up to date, it lacks the organization and level of abstraction necessary for effectively classifying vulnerabilities.

According to another study conducted by Sergei et al [34], vulnerabilities are classified into security, functional, operational, and developmental issues. By security issues, we mean flaws that can be abused to gain access to a user's data or assets. By functional issues, we mean flaws that result from an incorrect implementation of the business logic in the smart contract. By operational issues we refer to vulnerabilities that result from the incorrect operation of the smart contract, for example, if a contract is intended to be used for a limited period and the contract is not programmed to automatically close after that period then we have an operational issue. Finally, developmental issues are easily remedied vulnerabilities that are typically the result of poor coding practices.

The Decentralized Appliaction Security Project (DASP) Top 10 taxonomy developed by the NCC group was created with the aim of providing a high-level overview of the most common smart contract vulnerabilities [35]. The taxonomy is based on the OWASP Top 10 taxonomy used to classify the most common web application security vulnerabilities. The DASP Top 10 also contains a list of updated issues affecting smart contracts.

3.3 The DASP Top 10

In this section, we present the DASP Top 10 and discuss some of the most common vulnerabilities affecting smart contracts.

3.3.1 Reentrancy

Reentrancy is a flaw in smart contracts that allows an attacker to repeatedly call a function in order to trigger the execution of multiple transactions in a single transaction. Simply put, reentrancy attacks occur when an attacker reenters a smart contract and repeats the same action prior to the contract's state being updated. The DAO hack, which is regarded as one of the earliest smart contracts exploits on the Ethereum blockchain, occurred as a result of a reentrancy attack.

When a contract is vulnerable to a reentrancy attack, an attacker can launch an attack by invoking the vulnerable function and passing a malicious contract as a parameter. After the original contract updates its state, the malicious contract is programmed to execute its own code and then call the vulnerable function again. The attacker can repeat this action until the transaction runs out of gas or until the attacker's specified endpoint is reached. Consider the example contract in Listing 3.1, The ***Donation*** contract enables anyone to donate any amount of Ether to the smart contract, the *deposit()* function in line 10, enables users to deposit Ether to the contract and then updates the contract balance, the *withdraw()* function enables users to withdraw the Ether deposited by the user. In the *withdraw()* function, the Ether is returned to the user before the contract balance is deducted.

```
 1 pragma solidity ^0.4.24;
 2 contract Donation {
 3     uint balance;
 4     address owner;
 5     constructor() public {
 6         owner = msg.sender;
 7         balance = 0;
 8     }
 9
10     function deposit() public payable {
11         require(msg.value > 0);
12         balance += msg.value;
13     }
14     function withdraw(uint _amount) public {
15         require(msg.sender == owner);
16          msg.sender.call.value(_amount)();
17          balance -= amount;
18     }
19 }
```

Listing 3.1. Contract vulnerable to reentrancy attack.

In an attack scenario, a malicious contract can be deployed that invokes a fallback function which subsequently makes invokes the *withdraw()* function of the *Donation* smart contract, Listing 3.2 illustrates an example of such a malicious contract.

```
 1 pragma solidity ^0.4.24;
 2 
 3 contract ReentranceAttack {
 4     Donation public donation;
 5 
 6     constructor(address _donationAddress) {
 7         donation = Donation(_donationAddress);
 8     }
 9 
10     function() external payable {
11         if(address(donation).balance >= 1 ether) {
12             donation.withdraw(1);
13         }
14     }
15 
16     function attack() external payable {
17         donation.deposit{value: 1 ether}();
18         donation.withdraw(1);
19     }
20 
21 }
```

Listing 3.2. Reentrancy malicious contract.

To exploit the donation contract the attacker would call *ReentranceAttack.attack()* to donate 1 Ether to the *Donation* contract, the attacker would immediately invoke a withdrawal of 1 Ether from the *Donation* contract in line 18, this triggers the reentrancy as the *Donation.withdraw()* function invokes *ReentranceAttack* fallback function in line 10, and triggers a withdrawal loop; withdrawals will continue until the transaction gas is exhausted or the donation contract balance falls below 1. Figure 4 illustrates the iterative process.

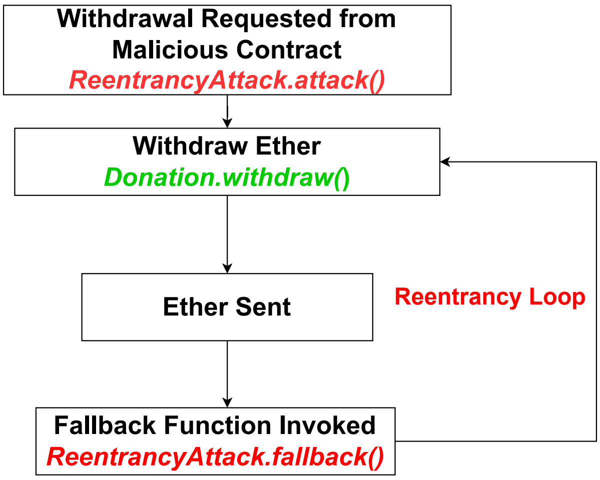

Withdrawal Requested from Malicious Contract
ReentrancyAttack.attack()
Withdraw Ether
Donation.withdraw()
Ether Sent
Reentrancy Loop
Fallback Function Invoked
ReentrancyAttack.fallback()

Figure 4. Recursive loop of a reentrancy attack.

There are several mitigations for reentrancy attacks. A common mitigation is to use a `mutex`, which is a type of lock that is used to ensure that only one thread can execute a critical section of code at a time. In Listing 3.3, a mutex is applied to patch the reentrancy vulnerabilities of the Donation contract. When the contract is invoked for the first time, a variable `lock` is set to `true`, and a check is implemented in line 12; if the contract is invoked again while `lock` is still `true`, the withdrawal does not go through to the malicious contract.

```
1  pragma solidity ^0.4.24;
2  contract Donation {
3      uint balance;
4      address owner;
5      constructor() public {
6          owner = msg.sender;
7          balance = 0;
8      }
9
10     function deposit() public payable {
11         require(!lock);
12         lock = true;
13         require(msg.value > 0);
14         balance += msg.value;
15
16     }
17
18     function withdraw(uint _amount) public {
19          require(!lock);
20          lock = true;
21          require(msg.sender == owner);
22          msg.sender.call.value(_amount)();
```

```
        balance -= _amount;
        lock = false;
    }
}
```

Listing 3.3. Reentrancy mitigation via mutex

Another preventive measure is by using the Checks-effects-interactions pattern, which involves executing internal calls and updating the state before making any external calls. An example is show in Listing 3.4, the balance of the contract is updated in line 3 before the transfer is made to the 3rd party.

```
1 function withdraw(uint _amount) public {
2         require(msg.sender == owner);
3         balance -= _amount;
4         msg.sender.call.value(_amount)();
5     }
```

Listing 3.4. Check effects pattern mitigation.

3.3.2 Access Control

Access control is a type of security mechanism that restricts a user's access to resources based on the user's identity. When a user attempts to access a resource, the access control system checks the user's identity and authorization level to determine whether the user is allowed to access the resource. Access control is used in the context of smart contracts to restrict access to specific functions of the contract. Access control is a critical security mechanism that is used to prevent unauthorized access to a smart contract.

The two widely publicised Parity exploits were caused by an access control vulnerability, which resulted in financial losses in the first case [36] and an accidental lock of $280 Million worth of ETHER in the second case [37].

Consider the example in Listing 3.5. A conceptual ***Savings*** contract has been deployed which ideally should allow only the owner of the contract withdraw Ether from the smart contract, but since no access control is implemented anyone could call *withdraw()* and drain the contract.

```
pragma solidity ^0.4.24;
contract Savings {
    address owner;
    uint balance;

    constructor() public {
        owner = msg.sender;
        balance = 0;
    }

    function deposit() public payable {
        require(msg.value > 0);
        balance += msg.value;
    }

    function withdraw(uint amount) public {
        balance -= amount;
        msg.sender.call.value(amount)();
    }
}
```

Listing 3.5. Contract vulnerable to access control bypass.

Modifiers can be used to implement access control in smart contracts. Modifiers are typically used to restrict access to certain functions of a smart contract. A function modifier can be added to the Savings contract to grant only the owner or deployer of the contract access to withdraw the Ether in the contract, Listing 3.6 shows an example of a function Modifier in solidity. The "_; " in line 3 serves as a check, if the owner invokes the contract then it runs without issues, else it throws an exception.

```
1 modifier onlyOwner() {
2         require(msg.sender == owner);
3         _;
4     }
```

Listing 3.6. A function modifier in solidity [38].

The Modifier can then be appended to any function, In the case of the vulnerable ***Savings*** contract in Listing 3.5, the modifier is applied to the *withdraw()* contract and shown in Listing 3.7.

```
function withdraw(uint _amount) public onlyOwner {

        require(msg.sender == owner);
        balance -= _amount;
        msg.sender.call.value(_amount)();
    }
```

Listing 3.7. A modifier applied to a vulnerable function.

3.3.3 Arithmetic Issues

Arithmetic issues occur when a smart contract performs unsafe arithmetic operations on integers. Due to the general lack of support for floating-point values in smart contracts, integers are used to represent values. Generally, arithmetic issues are referred to as integer overflows and underflows [38].

Listing 3.8 illustrates an example of integer overflow. The ***Timelock*** contract is intended to allow users to deposit and lock their Ether for a specified period of time. The user can also extend the `locktime` by invoking *increaseLockTime()* and specifying the number of seconds to add to the lock. If by any chance an attacker is able to obtain the private key to the user wallet, they can call the withdraw function regardless of the lock time due to a lack of integer overflow checks, to exploit this an attacker will first query the current `locktime` from the contract and then pass an argument

$$2^{256} - locktime$$

since the maximum value for an unsigned integer(uint) is

$$2^{256} - 1$$

this triggers an overflow and returns 0, which resets the `locktime` of the contract to 0, allowing the attacker to take all the funds stored in the contract.

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        uint transferValue = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

Listing 3.8. A Vulnerable smart contract with integer overflow issues [39].

An effective way to prevent arithmetic issues is by making use of trusted Maths libraries during development. `SafeMath` is a popular maths library designed by Openzeppelin to address this vulnerability. Listing 3.9 shows how the *add()* function from the `SafeMath` library is used to address the vulnerability in the ***TimeLock*** contract.

```
library SafeMath {
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

contract TimeLock {
    using SafeMath for uint;
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;

    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] = lockTime[msg.sender].add(
            _secondsToIncrease);
    }
}
```

Listing 3.9. Addressing overflow issues using SafeMath [39].

Real world examples of exploits due to arithmetic issues include the proof of weak hands exploit (PoWHC) [39] and the batch transfer overflow vulnerability (CVE-2018-10299) [40].

3.3.4 Unchecked Return Values For Low Level Calls

Low level calls are those that are made to the internal functions of a smart contract. `call()`, `callcode()`, `delegatecall()` and `send()` are all low-level calls in Solidity. Unchecked return values refer to the practice of not verifying a low level call's return value prior to using it. This can result in vulnerabilities if the low level call's return value is not as expected. For instance, if a contract calls the `send()` function of an external contract without checking the return value, the contract may be vulnerable to a reentrancy attack. It is usually advised to avoid low level calls entirely. Consider the example Lottery contract in Listing 3.10 , if for some reasons *‘winner.send()‘* fails the, the payedOut variable is set to true, effectively locking the prize money in the contract, since no checks are implemented to verify the return value of *‘winner.send()‘*.

```
contract Lottery {

    bool public paid = false;
    address public winner;
    uint public prize;

    function sendPrize() public {
        require(!payedOut);
        winner.send(10);
        payedOut = true;
    }

}
```

Listing 3.10. Unchecked low level calls

In the case that a low level call cannot be avoided it is essential to check the return value to handle any possible exceptions. A mitigation is shown in Listing 3.11 by checking for the return value of send() before setting payedOut to true, this way the win amount never gets locked in the contract.

```
contract Lottery {

    bool public paid = false;
    address public winner;
    uint public prize;

    function sendPrize() public {
        require(!payedOut);
        if (winner.send(10)) {
            payedOut = true;
        }
        else {
            throw;
        }
    }
}
```

Listing 3.11. Implementing checks for low level calls

A popular real world example is the *King of Ether* exploit in 2016 [41].

3.3.5 Denial of Service

Denial of service (DoS) attacks are a type of attack that attempts to make a system or network resource unavailable to its intended users. DoS attacks are usually launched by flooding the target system with requests that consume all the system's resources. A DoS attack, in the context of smart contracts, is a type of attack that aims to render a smart contract inaccessible to its intended users. The attack surface for DoS vulnerabilities is large and can be triggered unintentionally as a result of poor coding practices; the second parity hack is a real-world example of how a DoS attack was triggered unintentionally, resulting in the permanent locking of $280 Million in the smart contract [36].

Listing 3.12 illustrates a vulnerable smart contract that allows users to invest a specified amount of Ether and receive back five times the amount invested. An attacker can choose to invest using multiple wallets, flooding the investors array in the process, and can continue doing so until the gas required to execute the *distribute()* function exceeds the block gas limit [39].

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of
               tokens to the address "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Listing 3.12. Smartcontract vulnerable to DoS [39].

A suitable mitigation for this issue would be to alter the distribution logic and add logic that enables users to withdraw Tokens independently of the forloop.

3.3.6 Bad Randomness

Bad randomness is a vulnerability that allows an attacker to predict the output of a random number generator. This can be exploited to manipulate the outcome of a contract. For example, consider a contract that generates a random number and then uses the random number to determine the winner. If an attacker can predict the output of the random number generator, the attacker can manipulate the outcome of the game.

Numerous applications on the Ethereum blockchain require randomness, ranging from lottery decentralized applications to gambling and some financial decentralized applications. However, because everything on the blockchain is designed to be publicly visible, generating a true random number is challenging.

The source of randomness in blockchain should be external to the blockchain, one way of mitigating the issue of bad randomness is by using external oracles and entities as a source of truth. The CryptoPuppies Dapp was exploited as a result of bad randomness [42].

3.3.7 Front Running

Front Running is a type of attack that involves an attacker observing a transaction and then executing a transaction that is dependent on the original transaction. Front running attacks are common in decentralized exchanges.

Ethereum utilizes a Proof of Work (PoW) consensus algorithm, which means that miners are in charge of validating and adding transactions to the blockchain. Miners are compensated for their efforts through a fee that is paid by the transaction's sender. An attacker can monitor pending transactions. Consider Listing 3.13, in which the first person to solve the puzzle wins 1000 ether. The attacker could simply monitor the pending transaction pool, obtain the solution, and then submit the transaction with a higher gas fee. The miner will then accept the transaction with the highest gas fee, thereby awarding the prize money to the attacker.

```
contract FindThisHash {
    bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Listing 3.13. Simple puzzle contract [39].

Mitigating this vulnerability is difficult because it is influenced by two factors: users and miners. Miners can reorder and process transactions as they see fit, but their use as an attack vector is typically limited because they cannot predict when they will mine a block.

On the other hand for users, a gas threshold can be implemented in the smart contract to prevent front running. Front running attacks are popular on the major DEX on the Ethereum blockchain [43].

3.3.8 Time Manipulation

Time manipulation also known as *Timestamp Dependence* or *Time Constraints* is a type of attack that involves a miner manipulating the time of a transaction in order to exploit a vulnerability in a smart contract [38]. Time manipulation attacks are possible in smart contracts that use the `block.timestamp` variable for generating random numbers. The `block.timestamp` variable is a timestamp that is associated with each block in the Ethereum blockchain. The timestamp is used to determine when a transaction was mined. Consider Listing 3.14, the miner can manipulate the timestamp to fit the condition in line 9

10 in order to receive the Ether reward.

```
pragma solidity ^0.4.24;

contract Play {
     uint public pastTime;
     constructor() payable {}

     function play() external payable {
        require(msg.value == 10 ether);
        require(block.timestamp >= pastTime);
        pastTime = block.timestamp;
    // if pastTime is divisible by 9 you win the Game
        if(block.timestamp % 9 == 0) {
          msg.sender.transfer(1500 ether);
        }
     }
 }
```

Listing 3.14. Vulnerable to time manipulation.

To mitigate this vulnerability blockchain timestamp should not be used to generate entropy or random numbers. A real world example is the GovernMental exploit [39].

3.3.9 Short Address Attack

Short Address is a type of vulnerability that occurs as a result of the EVM accepting incorrect arguments, for example, the Ethereum Virtual Machine (EVM) uses 20-byte addresses to identify accounts. When a transaction is sent to an address that is shorter than 20 bytes, the EVM will pad the address with zeros. This can result in the loss of funds if the address is padded with zeros that were not originally part of it. To mitigate this vulnerability, it is essential to validate input on the clients side before sending the inputs to the blockchain.

3.3.10 Unknown Unknowns

Unknown unknowns are vulnerabilities that have not been discovered or disclosed. Unknown unknowns are difficult to mitigate, as there is no known way to prevent or detect them. Unknown unknowns can only be mitigated by continual auditing of smart contracts.

3.4 Discussion

To begin, Section 3.2 conducts a state-of-the-art evaluation of existing research on smart contract vulnerability classification; we compare the classification provided in each study to the DASP Top 10, and we discuss the limitations of past research.

As seen in Section 3.3, Solidity is prone to errors, and even minor errors can have catastrophic repercussions, resulting in large financial loss. Furthermore the simplified versions of real life exploit scenarios provided for each vulnerability, shows how easy it is to exploit some of the vulnerabilities. The common vulnerabilities discussed in Section 3.3 aligns with previous research done by Naoma et al [44], although the research by Naoma et al covers 8 classes of vulnerabilities mentioned in section 3.3, attack techniques such as front running and time manipulation are not covered. In another study by Huashen et al [45], similar results are produced but the vulnerabilities shown in section 3.3, are divided into additional subgroups, For example Denial of Service is broken into two subgroups namely, Frozen Ether and Unprotected Suicide [45].

3.5 Conclusion

This chapter was focused on identifying the common smart contract vulnerabilities on the Ethereum Blockchain, previous research work done on Smart Contract Vulnerability taxonomy domain were analysed in Section 3.2, and the process of selecting a suitable taxonomy was also explained.

The research question **What are the common vulnerabilities affecting smart contracts on the Ethereum blockchain?** has been answered using the DASP Top 10 taxonomy, Common vulnerabilities on the Ethereum blockchain include **Reentrancy** attacks a vulnerability which is caused as a result of a smart contract making insecure external calls, this also tops the list of attack vectors as seen in [44], [45], **Access Control** vulnerability as a result of improper access checks, **Arithmetic vulnerabilities** which refer to flaws introduced as a result unsafe arithmetic operations, **Unchecked Return Values for Low Level Calls** due to inappropriate calls to low level functions such as `call(), callcode()` `delegatecall()` and `send()`, **Denial of Service (DoS)** attacks as a result of poor coding practices, **Bad Randomness** which refers to vulnerabilities as a result of improperly generated random numbers, **Front running** vulnerabilities as a result of miners taking advantage of pending transactions in the mempool, **Time Manipulation** as a result of depending on `block timestamp` which can be manipulated by miners, **Short Address** attack as a result of improper user input validation and finally **Unknown Unknowns** which are vulnerabilities not discovered or known yet.

There are a number of limitations in our approach. First, our approach only identifies a subset of all possible vulnerabilities. Second, our approach only assesses a subset of all possible exploit examples. Finally, our approach is limited to Solidity, and therefore is not applicable to smart contracts written in other languages. In the future, we plan to expand our approach to identify more vulnerabilities, We also plan to extend our approach to smart contracts written in other languages.

4. Evaluation Requirements for Smart Contract Vulnerability Detection Tools on the Ethereum blockchain

The following chapter presents a utility evaluation process for smart contract vulnerability detection tools. The requirement specification of smart contract vulnerability detection tools are presented, furthermore, a scoring guideline for each requirement is provided, and a utility evaluation equation is presented. Finally, the evaluation process is deconstructed and discussed in detail.

4.1 Introduction

The objective of this section is to answer the Research Question RQ2 - **What are the evaluation requirements for vulnerability detection tools on the Ethereum blockchain?**

In Section 4.2, we will establish the requirements specification of a vulnerability detection tool and also provide a guideline on how to assign scores to each requirement. Section 4.3 will explore how to evaluate the utility of the vulnerability detection tools.

4.2 Requirement Specification of a Smart Contract Vulnerability Detection Tool

The bedrock of vulnerability detection tools are the functionalities they offer, we explore some of the important features of smart contract vulnerability detection tools, within the context of Ethereum smart contracts.

4.2.1 Functional Requirement

There are different kinds of functional requirements that a smart contract vulnerability detection tool should meet.

a. Method of Analysis

Vulnerability detection should involve at least one security testing methodology, smart contract security testing methodologies include:

i. Static Analysis: Static code analysis is the process of analyzing code without executing it. Smart contract analysis can be beneficial for identifying potential vulnerabilities in smart contracts as well as for understanding how the code works. Static analysis enables automated security reviews and may run more quickly on large contract files, it is also believed to be a cost-effective method of discovering vulnerabilities, as smart contracts cannot be modified once they are deployed on the blockchain. There are usually three stages in static analysis [46].

1. Building an intermediate representation
2. Enrichment of Intermediate Representation using algorithms such as symbolic execution and abstract interpretation
3. Vulnerability detection

ii. Dynamic Analysis: Dynamic analysis is the analysis of code during the execution of the code. Dynamic analysis has the advantage of identifying issues that may not be identified by static analysis, since static analysis can not identify every possible execution path of the code. However, dynamic analysis is a more expensive approach to security testing than static analysis.

b. Level of Abstraction

Vulnerability analysis should begin either from the Solidity source code or bytecode. The bytecode of contracts is the preferred option due to the lack of formal semantics in solidity and behavioural change across different compiler versions.

c. Bulk Analysis Support

Vulnerability detection tools should also provide support for bulk analysis, for example, tools should support detection of vulnerabilities for smart contracts already deployed on the Ethereum blockchain. Bulk analysis is important for analysing smart contracts in a distributed manner.

d. Detection Methodology

Vulnerability detection methodology is an important aspect of vulnerability detection tools. The various methods for vulnerability detection include:

i. Code Instrumentation: Code instrumentation is a method of modifying code to insert hooks or tracers in order to monitor code execution. Code instrumentation is used in the context of security testing to check assertions and monitor performance of the contract under analysis.

ii. Symbolic Execution: Symbolic execution is a technique for analysing programmes in which logical formulas representing the program's execution are created and solved. In the context of smart contract security testing, symbolic execution is a technique that utilises constraint solvers to attempt to calculate a concrete input by exploring all possible execution paths in the code [47].

iii. Constraint Solving: Constraint solving is the process of solving logical formulas by assigning values to variables such that the formula is true. Constraint solving is used in the context of smart contract security testing to find input values that satisfy a logical formula.

iv. Abstract interpretation: Abstract interpretation is a program analysis technique where a program is analyzed by calculating abstractions of the program instead of analyzing the concrete program. Abstract interpretation is used in the context of smart contract security testing for performing data flow analysis of smart contracts

v. Model Checking: Model checking is a formal verification technique where a model of a system is checked against requirements. Model checking is used in the context of smart contract security testing to automatically verify properties of smart contracts.

Vulnerability detection tools need to be able to detect vulnerabilities using one of the methods discussed above

e. Code Transformation Support

Code transformation is used in the context of security testing to improve the performance and accuracy of code analysis. Code transformation can be used to improve the accuracy of programme analysis by converting the code to a more easily-analyzed format. Vulnerability detection tools employ a variety of code transformation techniques, including the following [48]:

i. Disassembly: Disassembly is a technique used in transforming the Smart contract's bytecode into human readable instructions. Disassembly is used in the context of smart contract security testing to improve the accuracy of smart contract code analysis.

ii. Decompilation: Decompilation is the process of reverse engineering compiled code, this is important in order to provide contextual feedback on vulnerabilities. Decompilation is used in the context of smart contracts to convert EVM bytecode to a higher abstraction level, such as solidity code, in order to improve code readability and data flow analysis.

iii. Abstract Syntax Tree: This refers to a tree representation of the code, it is used to represent the structure of the code and can be used for various purposes such as data flow analysis, verification and validation. Smart contract security testing tools use Abstract Syntax Tree for the purpose of data flow analysis.

iv. Control flow graph: A control flow graph is a graph representation of the code that is used to highlight the paths that may be traversed during code execution [49].

v. Contextualization Support: This refers to the feedback of where an issue was detected in the solidity code or bytecode of a contract, a vulnerability detection tool should be able to point to the vulnerable part of the code either by specifying the line in the code or the problematic function [48].

g. Vulnerability Detection Support

Vulnerability detection tools should also be able to detect security issues in smart contracts, using the Dasp Top 10 as a reference [35], some of the vulnerabilities that should be detected by a smart contract vulnerability detection tool include:

i. Reentrancy (RE): Reentrancy is a type of vulnerability that allows an attacker to call a function of a smart contract recursively. This type of vulnerability can be used to exploit vulnerabilities in other smart contracts that use the contract with the RE vulnerability.

ii. Access Control: Access control is the process of limiting a resource's accessibility. Access control can be used to restrict access to certain functionality within a smart contract, such as restricting the ability to withdraw funds to the contract owner only, when access control is not implemented correctly it can lead to devastating consequences.

iii. Arithmetic Issues: Arithmetic issues occur when a smart contract performs unsafe arithmetic operations on integers.

iv. Unchecked Return Values For Low Level Calls: Unchecked return values can occur when a smart contract does not check the return value of a low level call. This can lead to vulnerabilities in the smart contract.

v. Denial of Service: Denial of Service (DoS) is a type of attack in which legitimate users are denied access to a resource. This is a type of vulnerability in which the smart contract is overburdened with computations that take a long time to complete.

vi. Bad Randomness Bad randomness is an issue that can occur when a smart contract uses an insecure source of randomness.

vii. Front Running Front Running is a type of attack in which an attacker observes a transaction and then executes another transaction that is dependent on the original.

viii. Timestamp Manipulation: Timestamp manipulation is a type of vulnerability in smart contracts that occurs when the value of a timestamp is used to determine the contract's behaviour. Dependence on timestamps can be used to exploit flaws in smart contracts.

ix. Short Address Attack: Short Address Attack is a type of vulnerability that occurs as a result of lack of input validation, It can also be as a result of EVM accepting improperly padded arguments.

4.2.2 Non-functional Requirement

For smart contract vulnerability tools, Non-functional requirements refers to the attributes of the tool that are important, but are not related to the functionality of the tool, these may refer to attributes that cannot be out-rightly observed during execution of the tool. The non-functional requirements considered in this thesis include:

h. Integrability

Integrability refers to the ability of a tool to be integrated with other tools, this is important for the purpose of security testing. For example, a security testing tool may be integrated with a testing framework to enable automated testing of smart contracts, or a security testing tool may be integrated into the CI/CD process in order to discover vulnerabilities during development.

g. Robustness

Robustness refers to a tool's ability to gracefully handle errors and cope with erroneous input. How frequently a tool fails to decompile or parse a smart contract without errors is important for security testing purposes. Security tools should be able to point out when an error has occured in the smart contract and in some cases the line in which the error occurs.

i. Usability

Usability refers to how easy it is to make use of a vulnerability tool. Vulnerability tools should have clear usage instructions and documentations. Interaction with the tool should be made easy using either a CLI, GUI or API.

j. Ease of Setup

Ease of setup refers to the ease with which a tool can be installed and configured by a user, as well as the ease with which users can utilize the said security tool. This is important for the purpose of security testing because if a tool is difficult to set up, users may be dissuaded from testing their smart contracts for vulnerabilities.

k. Flexibility

Flexibility refers to a tool's ability to be extended and customized by users. It also refers to a tool's ability to adapt to changes and customization's without compromising its utility, this is a desirable characteristic for tools that detect smart contract vulnerability.

4.3 Utility Evaluation of Smart Contract Vulnerability Detection Tools

Utility evaluation is the process of assessing the usefulness of a tool within a specific context, Utility is a function of both functional and non-functional requirements. Figure 6 depicts how utility is at the root of all functional and non functional requirements. Below, we propose a utility equation using the requirements specifications in Section 4.2.

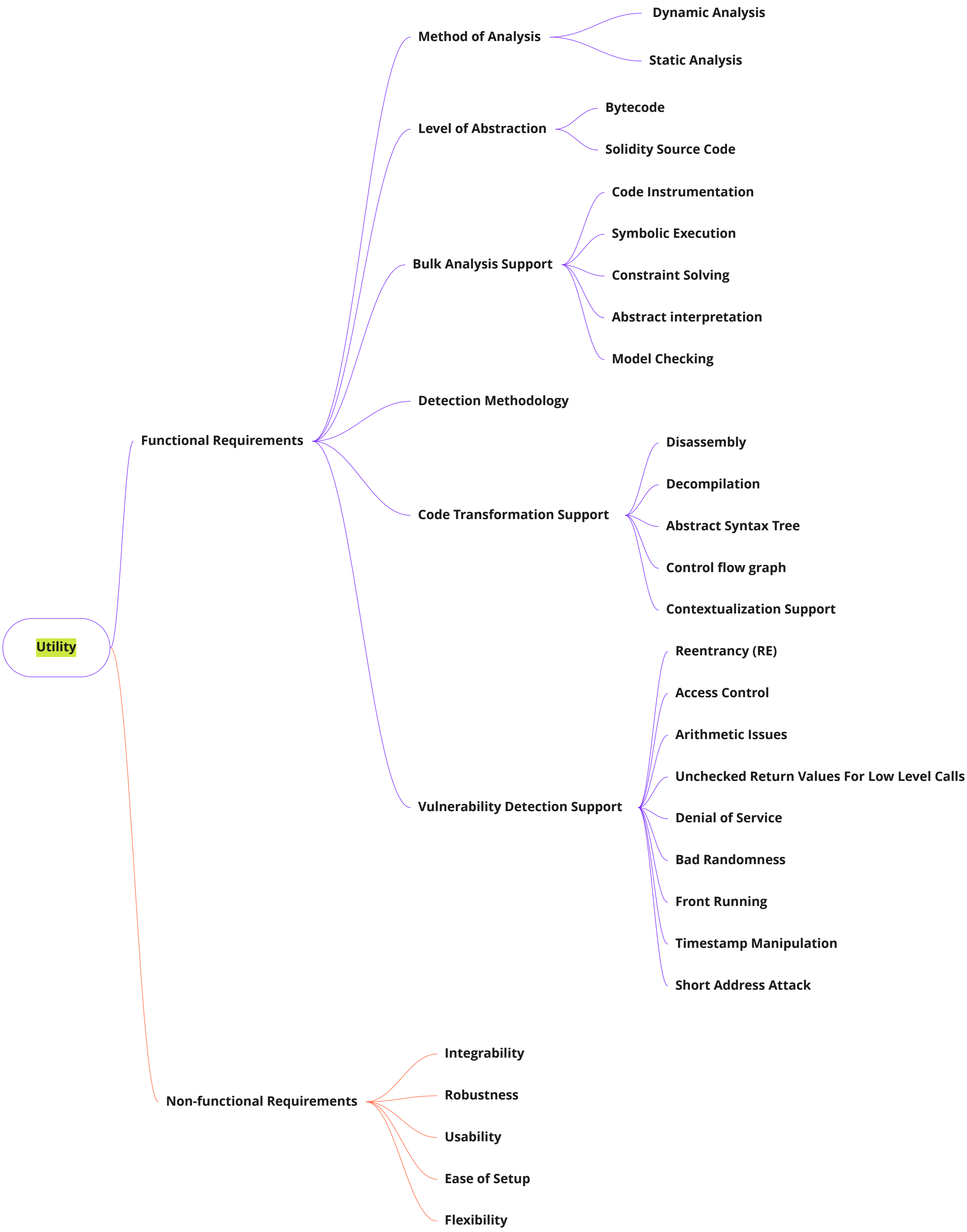


Figure 5. Requirements Specification Hierarchy

4.3.1 Utility Equation

For computing the utility *u* of a tool *t*, we consider Equation 4.1, in which each requirement r receives a score on the scale of {**3|2|1|0**}. Where 3 means a vulnerability detection tool has an **excellent support** for a requirement, 2 means **good support**, 1 means **minimal support** and 0 means a requirement is **not applicable** for the specific vulnerability detection tool. A scoring guideline for each requirement **Scores$_\mathbf{i}$** is presented in Table 4.

$$u^t = \sum_{i=1}^{r} Scores_i * Weight_i \tag{4.1}$$

Where:

r = Total number of requirements

Scores$_i$ = Score of a requirement

Weight$_i$ = Weight of a requirement

The **Weight$_\mathbf{i}$** refers to the weight of each requirement. This weight is used to compute the importance of each requirement within the context of a specific vulnerability detection tool.

4.3.2 Using the Utility Evaluation Framework

The utility of a tool can be determined using the methodology described here. Given a job J and a vulnerability detection tool T, the utility evaluation process entails allocating a weight W to all functional and nonfunctional requirements of the tool and then allocating a score to each requirement using the scoring guidelines in Table 4. The Three phases can be summarized as follows:

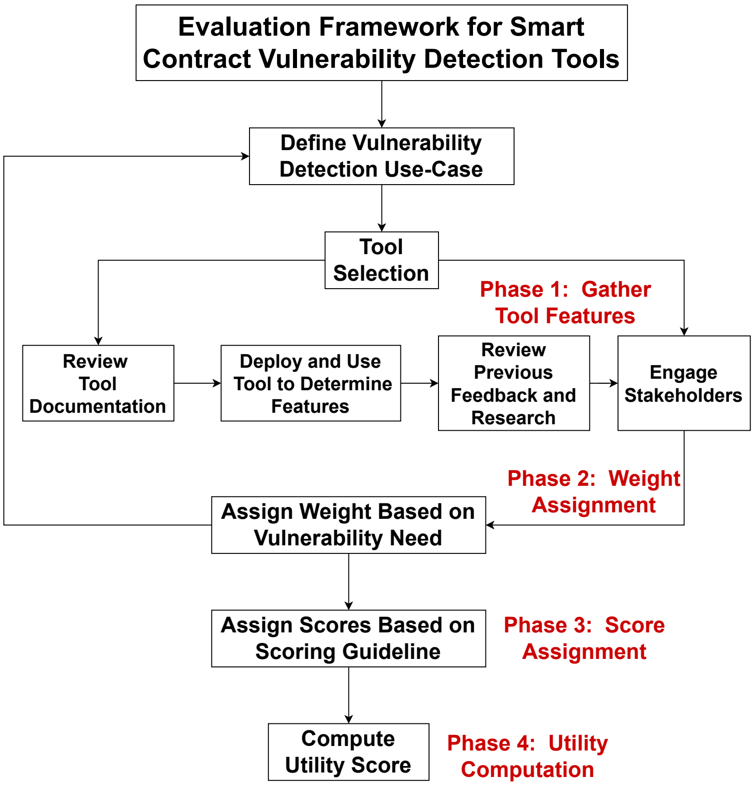

Evaluation Framework for Smart Contract Vulnerability Detection Tools
Define Vulnerability Detection Use-Case
Tool Selection
Phase 1: Gather Tool Features
Review Tool Documentation
Deploy and Use Tool to Determine Features
Review Previous Feedback and Research
Engage Stakeholders
Phase 2: Weight Assignment
Assign Weight Based on Vulnerability Need
Assign Scores Based on Scoring Guideline
Phase 3: Score Assignment
Compute Utility Score
Phase 4: Utility Computation

Figure 6. Utility Evaluation Framework

Phase 1: User outlines the respective features of the chosen smart contract vulnerability detection tool, this can be done by analyzing the tool creator's documentation or by running the tools themselves.

Phase 2: The user assigns a weight to each specification based on its relevance to the job at hand. Since two jobs may not require the same functionality or assign the same level of importance to each requirement, the weight may differ across various jobs, for instance, if a user is responsible for integrating vulnerability detection scans into the continuous integration pipeline for developing smart contracts, he or she may wish to prioritize the Integrability requirement over the Usability requirement. i.e $\{\mathbf{w_{integrability}} > \mathbf{w_{usability}}\}$

Phase 3: Users assign scores to each requirement using the scoring guideline in table 2 as a reference.

Phase 4: Users compute the utility of T to J by multiplying each score by the weight for each feature and then summing up all the values.

Based on this framework the utility of a tool T to a job J can be said to be: $u^t = \sum_{i=1}^{r} Scores_i * Weight_i$, using this, we can compare the utility of different tools for a job. Figure 6 depicts the steps and phases involved in the utility evaluation process.

	3 - Excellent Support	2 - Good Support	1 - Minimal Support	0 Not Applicable	Description
Method of Analysis	2 or more method of analysis	1 method of analysis		Not Applicable	Refers to the Security Testing methodology, Options include: 1. Static Analysis 2. Dynamic Analysis
Level of Abstraction	2 or more levels of abstraction	1 level of abstraction		Not Applicable	Indicates the starting point for vulnerability analysis, available options include: 1. Bytecode 2. Solidity source code
Bulk Analysis Support	Supports multithreading analysis in its entirety, including for smart contracts that have already been deployed on chain.	Limited support for multi-threading and vulnerability analysis for on chain smart contracts	Minimal Support for Bulk scanning	Not Applicable	Refers to support for multi thread scans and support for vulnerability detection for on chain smart contracts.
Detection Methodology	The tool supports 3 or more detection methods	Supports up to 2 detection methods	Supports 1 detection method	Not Applicable	Refers to the vulnerability methodology supported by the vulnerability tool. options include: 1. Code Instrumentation 2. Symbolic Execution 3. Constraint Solving 4. Abstract Interpretation 5. Model Checking
Code Transformation Support	The tool supports 3 or more code transformation methods.	Supports up to 2 code transformation methods	Supports 1 code transformation methods	Not Applicable	Refers to the code transformation technique used by the tool; available options include: 1. Disassembly 2. Decompilation 3. Abstract Syntax Tree 4. Control flow graph 5. Contextualization Support
Vulnerability Detection Support	Detects 7 or more type of vulnerabilities	Detects 4-6 types of vulnerabilities	Detects 1-3 types of vulnerabilities	Detects 0 – 1 type of vulnerability	Refers to the type of vulnerability that the tool will detect. According to the DASP top 10, some of the vulnerabilities include the following: 1. Reentrancy 2. Access Control 3. Arithmetic Issues 4. Unchecked Return Values 5. Denial of Service 6. Bad Randomness 7. Front Running 8. Timestamp Manipulation
Integrability	Tool can be easily integrated with other tools	Integrity is largely supported, with some caveats.	Integrability support is limited	Not Supported	Refers to the ability of a tool to be integrated with other tools
Robustness	Errors are properly handled and parsing and decompilation run flawlessly.	While errors are properly handled, decompilation and parsing occasionally fail.	In most cases, errors are not properly handled, and decompilation and parsing fail.	Not Applicable	Robustness refers to the ability of a tool to handle errors gracefully.
Usability	Supports CLI, GUI and API for user interaction.	Supports 2 Options	Supports either GUI CLI or API	Not Applicable	Usability refers to how easy it is to make use of a vulnerability tool.
Ease of Setup	Proper and Complete Installation instructions and documentations are provided	Simple to configure, with only a few details missing from the documentation	Installation is challenging, and there is a deficiency in the documentation.		Ease of setup refers to the ease with which a tool can be installed and configured by a user

Table 4. Requirement Scoring Guideline

4.4 Discussion

This chapter answers the Research question **What are the evaluation requirements for vulnerability detection tools on the Ethereum blockchain**? We conclude that to efficiently evaluate the utility of a tool, the features of the tool needs to be examined individually as functional and non functional requirements based on the vulnerability detection use-case for a particular job. The functional features that make up the overall utility are: *Method of analysis, level of abstraction, bulk analysis support, detection methodology, code transformation support,* and *vulnerability detection support.* The non-functional requirements refer to attributes of the tool that cannot be out-rightly observed during its execution, they include: *integrability, robustness, usability, ease of set up* and *flexibility of the tool.*

A scoring guideline that can be applied to these requirements where presented in Table 4. The scoring guideline serves as a reference document to assign and weigh the distinct functional and non-functional features of a vulnerability detection tool, each feature is assign a score on the scale of {**3|2|1|0**}. Where 3 means a vulnerability detection tool has an **excellent support** for a requirement, 2 means **good support**, 1 means **minimal support** and 0 means a requirement is **not applicable** for the specific vulnerability detection tool.

Nevertheless, the requirements presented and scoring guide provided are just guidelines to follow, a user may use a different set of requirements and assign various weights to each requirement based on their vulnerability detection use-case.

4.5 Conclusion

Chapter 4 presented a requirement specification for smart contract vulnerability detection tools based on previous research by Monika et al [48], Moona et al [50], and the author's experience. Additionally, Figure 6 was presented to further explain how the functional and non functional requirements of a tool makes up its utility. A utility tree was developed producing two branches from the tree root: Functional and Non-Functional requirements.

The Functional branch of the tree includes: *Method of analysis, level of abstraction, bulk analysis support, detection methodology, code transformation support,* and *vulnerability detection support.* The non-functional branch refer to attributes of the tool that cannot be out-rightly observed during its execution, they include: *integrability, robustness, usability, ease of set up* and *flexibility of the tool.*

Furthermore, a novel evaluation method to determine the utility of a smart contract vulnerability detection tools using the utility equation was presented in Section 4.3, the utility equation 4.1 is used to calculate the overall utility score by using the scores and individual weights assigned to each feature. Finally, we presented and provided usage instructions for the evaluation framework, and also developed Figure 6 which divides the utility evaluation process into 4 phases namely:

1. Phase 1: Gathering of Tool Features.
2. Phase 2: Assigning weight to features based on vulnerability detection needs.
3. Phase 3: Score assignment using the scoring guideline in Table 4.
4. Phase 4: Utility Computation using Equation 4.1.

Future work includes extending the requirement specifications highlighted in Section 4.2 to cover for more tool features that may be available.

5. Features of Existing Open-Source Vulnerability Detection Tools

The following chapter presents an overview of the existing smart contract vulnerability detection tools on the Ethereum blockchain, we present some of the major features of these tools, we also explore their capabilities, furthermore we compare these tools using categories such as the method of analysis, detection methodology, level of abstraction, usability, code transformation method and vulnerability type detected.

5.1 Introduction

The objective of this chapter is to answer the final research question RQ3 - **What are the features of the vulnerability detection tools on the Ethereum blockchain?**

In Section 5.2, we will establish a set of criterion for tools to be included in this research, Section 5.3 will provide an overview of the existing vulnerability detection tools on the Ethereum blockchain. In Section 5.4 we will compare each tools based on six distinct categories.

5.2 Smart Contract Vulnerability Detection Tools on the Ethereum Blockchain

Smart contract vulnerability detection tools are used to identify potential vulnerabilities in smart contracts developed on the Ethereum blockchain. These tools can be used at various stages of the software development cycle to identify vulnerabilities in smart contracts, including before, during, and after contract deployment. Smart contract vulnerability detection tools analyse smart contracts using a variety of techniques, including static code analysis, symbolic execution, dynamic analysis, and formal verification. Security analysis of smart contracts using these tools typically entails analysing source code, bytecode, and transaction traces, as well as the generation of a report containing the analysis results. The methodology used by each tool varies, as does the vulnerability detected by each tool. According to the research done by Rameder et al [51] there are over 140 smart contract vulnerability detection tools available, out of which 83 are open source tools.

5.2.1 Tool Selection Criteria

To gain a better understanding of the state of the art in terms of smart contract vulnerability detection tools, we first review the previous research done by Rameder et al [51] In the research done by Rameder et al [51], it was discovered that there are 140 different smart contract vulnerability detection tools available, To streamline this research, the following criteria for selecting the tools to be reviewed have been established:

- *Selection Criterion 1*: The tool is published or updated within the year 2021 - Present.
- *Selection Criterion 2*: The tool accepts either Solidity Code or Bytecode.
- *Selection Criterion 3*: The tool is able to detect security vulnerabilities.
- *Selection Criterion 4*: The tool is able to perform security analysis with only a single input of either a soldity sourcecode or bytecode.

After applying *Selection Criterion 1 (SC1)*, we are left with sixteen tools that have been recently updated or published between 2021 and the present. *Selection Criterion 2-4* are then used to refine the list further. Following the application of *Selection Criteria 1-4*, we identified five tools that satisfy each criterion; these tools are ***Conkas, Manticore, teEther, Mythril***, and ***Slither***. Table 5 provides an overview of the tools that meet each selection criteria and the final list of selected tools.

Selection Criteria	**Tool Reviewed**
Published or updated from 2021 - Present (IC1)	Conkas, Echidna, ETHBMC, Ethersplay, ETHIR,Gigahorse, GNNSCVulDetector, Kevm, Manticore, Smartbugs, Solidfi, SolidityCheck, teEther, Vertigo, Mythril, Slither
Solidity Code or Bytecode Input (IC2)	Conkas, Echidna, ETHBMC, Ethersplay, ETHIR,Gigahorse, GNNSCVulDetector, Kevm, Manticore, Smartbugs, Solidfi, SolidityCheck, teEther, Vertigo, Mythril, Slither
Able to detect security vulnerabilities (IC3)	Conkas, ETHBMC, GNNSCVulDetector, Manticore, SolidityCheck, teEther, Mythril, Slither
Requires only Sourcecode or Bytecode for analysis (IC4)	Conkas, Manticore, teEther, Mythril, Slither
Selected Tools	Conkas, Manticore, teEther, Mythril, Slither

Table 5. Tool Selection Criteria

5.3 Tool Overview

The following section provides an overview of the five tools selected for this research, the five tools selected for this research are ***Conkas, Manticore, teEther, Mythril*** **and** ***Slither***.

5.3.1 Conkas

Conkas is a static analysis smart contract that detects vulnerabilities by utilising the symbolic execution model. Conkas constructs a Control Flow Graph (CFG) using the Ryan Stortz-developed Rattle engine, which is then transformed to an Intermediate Representation (IR) and passed down to the symbolic execution engine, which then generates traces, which are passed down to the Conkas detector modules.

Conkas currently supports five vulnerability categories: Time Manipulation, Reentrancy, Arithmetic, Front-Running, and Unchecked Low Level Calls. Conkas can scan vulnerabilities in both EVM bytecode and Solidity code. It is critical to understand that, while Conkas supports Solidity source code, analysis is performed at the bytecode level. Conkas includes a Command Line Interface (CLI), additionally, it is possible to specify the vulnerability category to check for. Users can specify the depth and intensity of a scan.

Conkas's primary known limitation is its inability to detect smart contract dependency files, which means that it may be unable to effectively detect vulnerabilities in smart contract dependencies and library files, unless specifically specified by the User during the analysis procedure.

Conkas was created by researchers at Portugal's Instituto Superior T'ecnico and is available as an open source tool via GitHub [52].

Features

- Solidity Code and Bytecode input support.
- Command Line Interface Support.
- Support for custom vulnerability scanning where you can specify the type of vulnerability to search for.
- Support for custom modules.
- Support for EVM instructions.
- Unit test support.

5.3.2 Manticore

Manticore is a dynamic symbolic execution engine developed by Trail of Bits that can be used to analyse smart contracts for vulnerabilities, Manticore utilises the symbolic execution engine to generate traces of all possible execution paths in a smart contract; these traces are then passed to the Manticore core engine modules responsible for identifying vulnerabilities.

Manticore employs the SMT solver Z3 to discover unique computational paths, then identifies the input that will initiate that path, and finally records the executional traces. Manticore is able to identify vulnerabilities such as reentrancy, arithmetic vulnerabilities, and selfdestruct operations by analysing the recorded executional traces.

Manticore supports contracts written in Solidity and Vyper and also provides a CLI and Python API. Using the API it is also possible to perform custom analysis. Manticore is developed by *Trail of Bits* and is available as an open source project on Github.

Features

- Supports Solidity and Vyper programmed smart contracts.
- Support for analysis using symbolic inputs.
- Automatic test input generation.
- Supports Error identification and handling in Solidity contracts.
- Provides a Python API for easy integration with other tools.

5.3.3 TeEther

TeEther is a static symbolic executioner tool for automatically generating exploits for vulnerable smart contracts. The teEther Control Flow Graph (CFG) module decompiles the EVM bytecode and then constructs a CFG interpretation. The CFG is then scanned for state changes and critical instructions and passed to the exploit generation module, which generates exploits for the vulnerable paths. [53].

TeEther uses the SMT-solver Z3 to generate multi-transactional exploits, teEther deploys the specified smart contract to a private blockchain in order to test the generated exploit and also to prevent false positives [48]. Vulnerability analysis is performed on the bytecode level, although teEther accepts the Solidity code and then transforms that into bytecode before passing it down to the SMT-solver. TeEther focuses on transactional flaws that can result in a payout to arbitrary addresses.

TeEther neither supports a CLI or GUI, Users have to manually compile the Python code and input the contract to be scanned manually inorder to perform security analysis.

TeEther was developed by researchers from CISPA, Saarland University and it available online as an opensource tool on GitHub [53].

Features

- Exploit Generation Support.
- Solidty to EVM bytecode transformation support.
- EVM bytecode dissembly support.
- CFG plotting support.

5.3.4 Mythril

Mythril is an open source smart contract analysis tool developed by the company *ConsenSys*. Mythril analyses smart contracts using its symbolic execution engine, LASER-Ethereum. The Control Flow Graph is used by LASER to organise the programme states, each node in the graph represents a block of code and each node has a set of path formulas [54]. To compute exploits for each identified vulnerability, the SMT solver Z3 is used.

Mythril protects against a variety of vulnerabilities, including reentrancy, integer overflow/underflow, arithmetic, and unchecked low level [55]. Mythril accepts an Ethereum smart contract address or solidity code as input and converts it to bytecode. Mythril employs the SMT solver Z3 to identify bytecode vulnerabilities.

At the moment, Mythril only includes a Command Line Interface (CLI) and a Dissembler for bytecode and on-chain smart contracts. Mythril also supports blockchain exploration, which enables users to scan a smart contract automatically by providing Mythril with only the smart contract's address.

Mythril was developed by *ConsenSys* is available for use to smart contract developers on GitHub.

Features

- Concolic testing support.
- Supports bytecode, solidity code and smart contract address input.
- Disassembler for bytecode strings.
- Blockchain exploration support.

- Supports remote smart contract scanning.

5.3.5 Slither

Slither is a static analysis tool for detecting smart contracts vulnerabilities. It accomplishes this by "converting solidity smart" contract code to an "intermediate representation (IR) called SlithIR" [56]. Slither takes the smart contract's Solidity Abstract Syntax Tree (AST) from the solidity compiler and converts it to SlithIR, Slither's internal representation language. SlithIR identifies smartcontract vulnerabilities through the use of static single assessment (SSA).

Slither is capable of detecting a broad range of security vulnerabilities, including reentrancy, tx-origin, timestamp, and unchecked low level calls. Slither analyses vulnerabilities in solidity source code. It currently supports 76 vulnerability detectors and enables users to specify the type of vulnerability they want to monitor.

Slither currently provides a Python API and a command-line interface. The API enables the creation of custom analyses. Slither also includes code linting capabilities and is capable of determining the location of an error in a smart contract. Continuous integration and Truffle builds are also supported by Slither.

Slither was developed by *Trail of Bits* in conjunction with researchers from the *Northern Arizona University* and its available as an open source tool on GitHub.

Features

- Supports CI Integration.
- Custom analysis support.
- Python API support.
- Code linting support.
- Automated code optimization detection.

5.4 Tool Comparison

In this section we compare the tools in Section 5.3. The tools are compared based on the following categories:

- i. Method of analysis.
- ii. Detection Methodology.

- iii. Level of Abstraction.
- iv. Usability type.
- v. Code Transformation method.
- vi. Vulnerability type detected.

Table 6 shows the comparison between the selected tools in Section 5.3 using the categories mentioned above.

Tools	**Method**		**Detection**			**Level**		**Usability**			**Transformation Method**				
	Static Analysis	Dynamic Analysis	Symbolic Analysis	Constraint Solving	Data dependency analysis	Bytecode	Solidity Code	GUI	CLI	API	Disassembly	Decompilation	Abstract Syntax Tree	Control Flow Graph	Contextualization Support
Conkas	✓		✓	✓		✓	✓		✓		✓		✓	✓	
Manticore		✓	✓	✓		✓			✓	✓	✓				✓
teEther	✓	✓		✓		✓					✓			✓	
Mythril	✓		✓	✓		✓			✓		✓	✓		✓	
Slither	✓				✓		✓		✓	✓			✓	✓	✓

Table 6. Comparison of Vulnerability Detection Tools

i. Method of Analysis: This refers to the type of analysis that the tool performs; Conkas, Mythril, and Slither are all static analysis tools, whereas Manticore is a dynamic analysis tool. TeEther provides support for both static and dynamic analysis of smart contracts [48].

ii. Detection Methodology: This refers to how the tool detects vulnerabilities, Conkas, Mythril, Manticore all employ symbolic execution methodology and constraint solving to detect vulnerabilities, teEther employs only constraint solving for vulnerability discovery [48], Slither uses data dependency analysis using its static single assessment transformer engine.

iii. Level of Abstraction: This is the level of abstraction at which vulnerability analysis begins. Manticore, Mythril, and teEther all support bytecode abstractions [48], whereas Conkas supports both bytecode and solidity code abstractions [52]. Slither currently only supports the Solidity source code.

iv. Usability type: This refers to the level of usability supported by the tool, Manticore and Slither both support both a graphical user interface (GUI) and an API, whereas Conkas and

Mythril only support a command line interface. teEther does not include a command-line interface, graphical user interface, or API, instead, teEther requires users to manually modify the Python script that compiles each smart contract to scan [53].

v. Code Transformation Method: This is the method by which the tool transforms the code during the analysis process. Conkas supports code transformation methods such as disassembly, abstract syntax tree (AST), and control flow graph (CFG) [52]. Manticore is limited to disassembly and code contextualization, teEther is limited to disassembly and CFG, Mythril is limited to disassembly, decompilation, and CFG [48], and Slither is limited to AST, CFG, and code contextualization.

vi. Vulnerability Type Detected: This refers to the type of vulnerabilities detected by the tool. Table 7 shows the vulnerabilities detected by each tool based on the DASP top 10 vulnerability.

Tool	**Access Control**	**Arithmetic**	**Denial of Service**	**Front Running**	**Reentrancy**	**Time Manipulation**	**Unchecked Low Level Calls**
Conkas		✓		✓	✓	✓	✓
Manticore	✓	✓			✓	✓	✓
teEther	✓	✓					
Mythril	✓	✓		✓	✓		✓
Slither	✓		✓		✓	✓	✓

Table 7. Vulnerability Detected By Tool

Conkas can identify five distinct types of vulnerabilities (Arithmetic, Front Running, Reentrancy, Time Manipulation, and Unchecked low level calls) [52]. Manticore is capable of identifying five distinct types of vulnerabilities (Access Control, Arithmetic, Reentrancy, Time Manipulation and Unchecked low level calls) [48]. TeEther is capable of detecting three types of vulnerabilities (Access Control, Arithmetic, and Reentrancy) [53], as it is primarily focused on exploit generation for transactional vulnerabilities. Mythril is capable of identifying five distinct types of vulnerabilities (Access Control, Arithmetic, Front Running, Reentrancy, Unchecked Low Level Calls) [55]. Additionally, Slither is capable of detecting five distinct types of vulnerabilities (Access Control, Denial of Service, Reentrancy, Time Manipulation and Unchecked Low Level Calls) [38].

5.5 Discussion

This chapter answer the Research question RQ-3 **What are the features of Ethereum smart contract detection tools?** We can conclude that all the tools have a strong focus on the types of vulnerabilities that are listed in the DASP top 10 vulnerability taxonomy. It is also clear that the tools have a fair amount of overlap in terms of the type of vulnerabilities detected by each tool. It is also clear from the comparison that the tools are implemented using different methodologies, and each tool focuses on a slightly different area.

In terms of usability, there are many Ethereum smart contract detection tools that support a graphical user interface (GUI), command line interface (CLI), and/or application programming interface (API)(Conkas, Manticore, Mythril, Slither). However, there are also Ethereum smart contract detection tools that neither support a command line interface, graphical user interface, or API (teEther). In terms of the level of abstraction, most Ethereum smart contract detection tools support bytecode abstractions (Conkas, Manticore, teEther, Mythril). However, some Ethereum smart contract detection tools support only solidity code abstractions (Slither). While some tools support both Bytecode and Solidity code abstractions (Conkas). In terms of the code transformation method, the popular options used across multiple tools are Disassembly and CFG (Conkas, teEther, Mythril and Manticore). Only one tool Mythril supports code decompilation. Access control, arthimetic, reentrancy, time manipulation, and unchecked low level calls are the most frequently detected vulnerability types by the tools compared. Conkas, Manticre, teEther, and Mythril all discover a similar number of vulnerability types, however, teEther is only capable of detecting two types of vulnerabilities: Access Control and Arithmetic issues. The least discovered vulnerability type is Denial of Service, only Slither is able to discover DoS vulnerabilities across the multiple tools compared.

There are currently no tools that support all the DASP top 10 vulnerabilities, That said, it is clear that the field is evolving. We can conclude that the security of Ethereum smart contract is in the early stage of development, and many tools are not mature enough to provide a comprehensive security analysis

5.6 Conclusion

Chapter 5 presented an overview of smart contract vulnerability detection tools on the Ethereum blockchain, We began our research from the paper released by Rameder et al [51], in which 140 Ethereum smart contract vulnerability detection tools were analysed, to streamline the list we established four criteria for selecting the tools to be reviewed in this research, each criterion was based on factors such as the year in which each tool was published or last updated, the abstraction level for each tool, the purpose of each tool and the input type required by each tool to perform security analysis. After applying each criterion we were left with five tools: Conkas, Manticore, teEther, Mythril and Slither.

Next we highlighted the features of each tool and compared each tool based on six distinct categories: Method of analysis, Detection methodology, Level of Abstraction, Usability type, Code Transformation and Vulnerability type detected.

There are a number of limitations in our approach to reviewing smart contract analysis tools. One such limitation is that the criteria we used to select the tools may not have been the most effective, for instance, we only selected tools that were published or updated from 2021 - Present, this may have resulted in some tools that could have made our final cut being excluded from the review. Another limitation is that the tools we chose to review were all open-source tools, it is possible that some proprietary tools could have been more effective at detecting vulnerabilities in smart contracts, however, the amount of information available on proprietary tools is very limited, which made it difficult to compare proprietary tools in the same way we compared open-source tools.

For future work additional tools could be reviewed, and an increased emphasis could be placed on proprietary tools.

6. Evaluation

In the following chapter an evaluation of the proposed framework is performed using the design science research descriptive evaluation method. Section 6.2 provides an overview of the framework's real-world implementation in the form of a proof of concept (PoC), Finally, in Section 6.3, an evaluation of the PoC prototype is conducted using two constructed scenarios.

6.1 Introduction

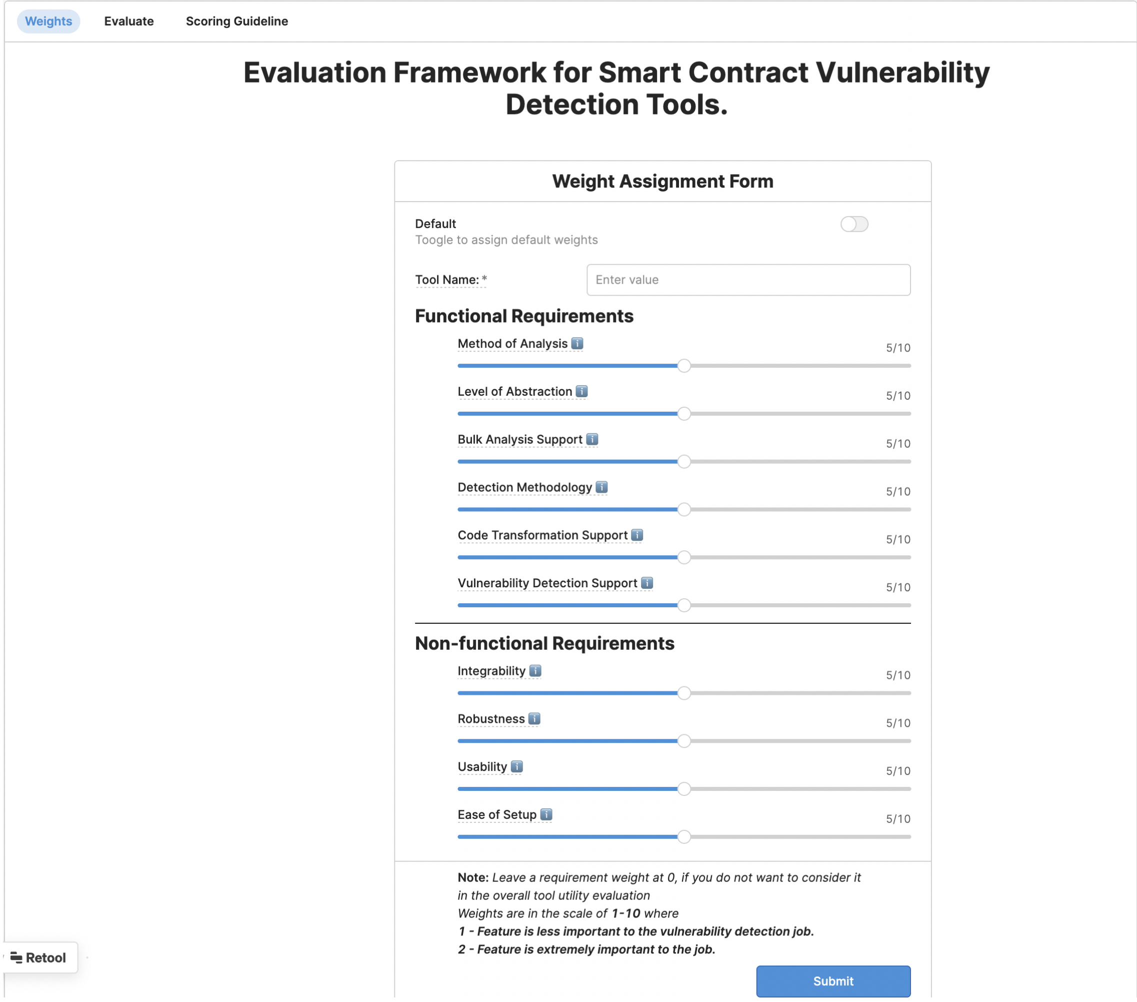
According to Hevner et al. [20], the evaluation process for design science research establishes a set of guidelines for presenting and justifying the design artifact. The *descriptive evaluation method* is the preferred method of evaluation for this study, it involves "*constructing detailed scenarios around the artifact to determine its utility*" [20]. Section 6.2 contains a proof of concept (PoC) implementation of the proposed framework for the purpose of evaluating it. The proof-of-concept implementation will be used to assess the framework's ability to meet the requirement of determining the utility of a vulnerability detection tool. Section 6.3 will evaluate the PoC implementation using two constructed scenarios. Both scenarios will be used to assess the framework's ability to accurately determine the utility of various vulnerability detection tools. The tools chosen for this research are ***Conkas, Manticore, teEther, Mythril, and Slither.***

6.2 Proof of Concept Implementation

The purpose of this Proof of Concept (PoC) is to develop a simple web application that smart contract developers, academic researchers, bug bounty hunters, and anyone else interested in evaluating the utility of smart contract vulnerability detection tools can use to evaluate their utility. The proof-of-concept prototype is implemented in two stages. To begin, the utility calculation is implemented in Javascript, a snippet of the code is included in Appendix 2. Second, a frontend is constructed using Retool, a low-code platform for the development of internal tools [57]. Retool was chosen as the platform for the PoC due to its simplicity of use and ability to quickly create forms and execute scripts.

Three (3) tabs are developed with retool, namely:

1. ***Weights Tab*:** This tab contains a weight assignment form, which contains a list of functional and non-functional tool specification requirements, as well as sliders ranging from 1 to 10. Using the sliders, users can adjust the weight assigned to each requirement based on the vulnerability detection use case. Hovering over each specification requirement also displays a definition of the requirements and outlines some of the options, for example hovering over the *level of abstraction* requirement, displays the definition and the available options such as *bytecode* and *solidity source code*. Figure 7 illustrates the Weights tab.



Evaluation Framework for Smart Contract Vulnerability Detection Tools.

Weight Assignment Form

Default
Toogle to assign default weights

Tool Name: * Enter value

Functional Requirements

Method of Analysis 5/10
Level of Abstraction 5/10
Bulk Analysis Support 5/10
Detection Methodology 5/10
Code Transformation Support 5/10
Vulnerability Detection Support 5/10

Non-functional Requirements

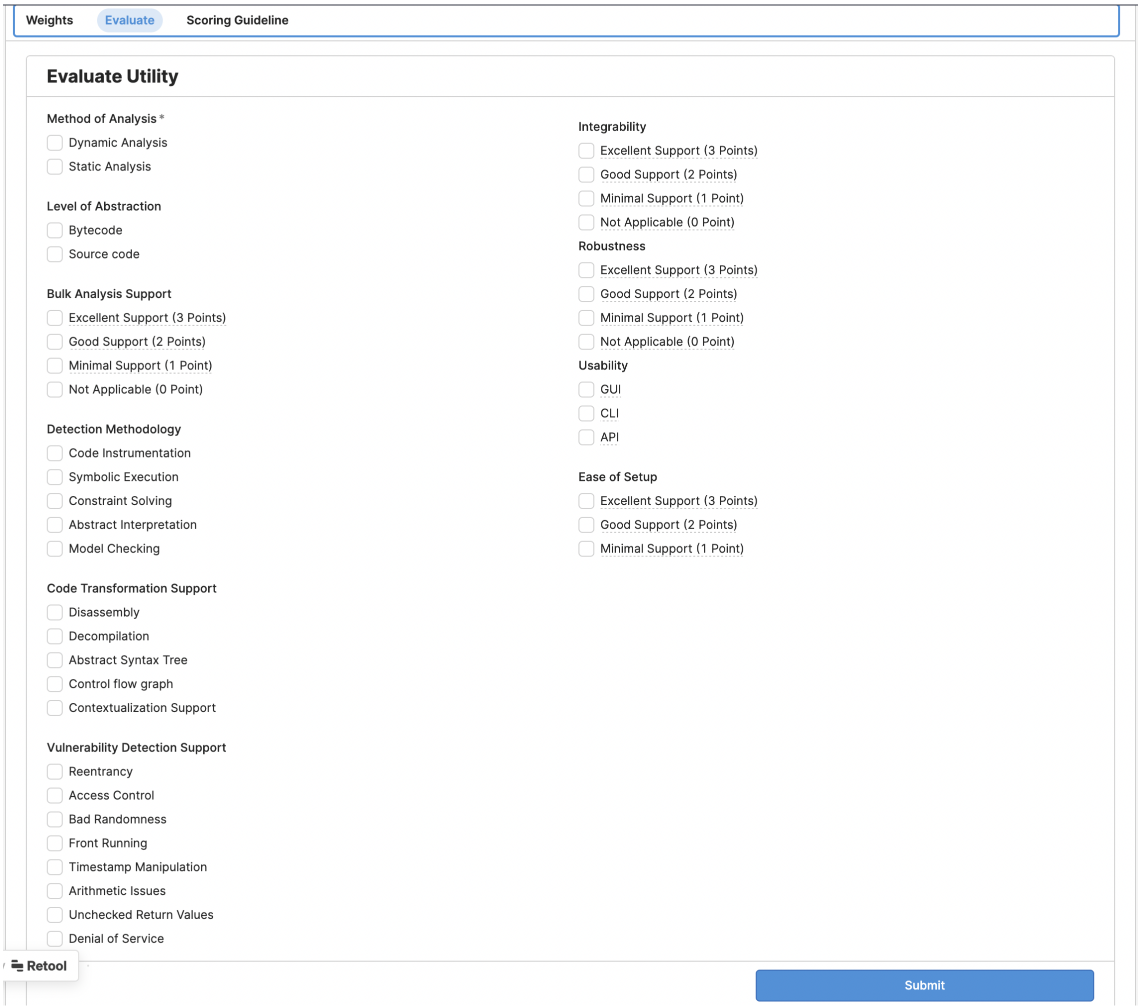
Integrability 5/10
Robustness 5/10
Usability 5/10
Ease of Setup 5/10

Note: *Leave a requirement weight at 0, if you do not want to consider it in the overall tool utility evaluation*
Weights are in the scale of ***1-10*** *where*
1 - Feature is less important to the vulnerability detection job.
2 - Feature is extremely important to the job.

Figure 7. Weights Tab.

When the user clicks the Submit button, the assigned weight is saved to the user's browser's local storage and the user is redirected to the Evaluation Tab.

2. ***Evaluate Tab***: The evaluation tab contains check boxes containing options for each specification requirement, the user selects each option that applies to the tool being evaluated. Figure 9 illustrates the evaluation tab.
When the user clicks on the *Submit* button, the overall utility of the tool is calculated



Evaluate Utility

Method of Analysis *

□ Dynamic Analysis

□ Static Analysis

Level of Abstraction

□ Bytecode

□ Source code

Bulk Analysis Support

□ Excellent Support (3 Points)

□ Good Support (2 Points)

□ Minimal Support (1 Point)

□ Not Applicable (0 Point)

Detection Methodology

□ Code Instrumentation

□ Symbolic Execution

□ Constraint Solving

□ Abstract Interpretation

□ Model Checking

Code Transformation Support

□ Disassembly

□ Decompilation

□ Abstract Syntax Tree

□ Control flow graph

□ Contextualization Support

Vulnerability Detection Support

□ Reentrancy

□ Access Control

□ Bad Randomness

□ Front Running

□ Timestamp Manipulation

□ Arithmetic Issues

□ Unchecked Return Values

□ Denial of Service

Integrability

□ Excellent Support (3 Points)

□ Good Support (2 Points)

□ Minimal Support (1 Point)

□ Not Applicable (0 Point)

Robustness

□ Excellent Support (3 Points)

□ Good Support (2 Points)

□ Minimal Support (1 Point)

□ Not Applicable (0 Point)

Usability

□ GUI

□ CLI

□ API

Ease of Setup

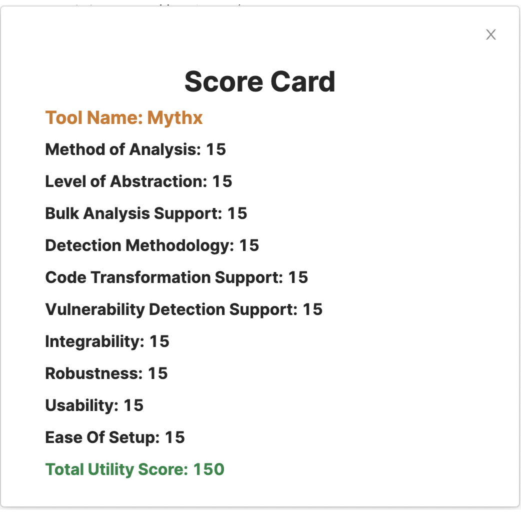
□ Excellent Support (3 Points)

□ Good Support (2 Points)

□ Minimal Support (1 Point)

Figure 8. Evaluate Tab.

using the evaluation framework described in Chapter 4, the result is then displayed as a modal, Figure 5 displays an example result.



Score Card

Tool Name: Mythx

Method of Analysis: 15

Level of Abstraction: 15

Bulk Analysis Support: 15

Detection Methodology: 15

Code Transformation Support: 15

Vulnerability Detection Support: 15

Integrability: 15

Robustness: 15

Usability: 15

Ease Of Setup: 15

Total Utility Score: 150

Figure 9. Score Card.

3. ***Scoring Guideline***: The Scoring guideline tab contains a scoring reference for each requirement specification, the PoC makes use of the scoring guideline displayed in Table 4.

The PoC is currently available at `https://cutt.ly/QGhliVa`

6.3 Prototype Evaluation

For evaluation of the prototype, the prototype will be used to evaluate five vulnerability detection tools based on the criteria outlined in Section 5.2.1 of this thesis. The tools selected are *Conkas, Manticore, teEther, Mythril* and *Slither*.

First the requirement specification options applicable or supported by each tool are outlined in Table 8.

Tools	Objective Criteria																						Subject Criteria			
	Method		Detection			Level		Usability			Transformation Method					Vulnerability Detected										
	Static Analysis	Dynamic Analysis	Symbolic Analysis	Constraint Solving	Data Dependency Analysis	Bytecode	Solidity Code	GUI	CLI	API	Disassembly	Decompilation	Abstract Syntax Tree	Control Flow Graph	Contextualization Support	Access Control	Arithmetic	Denial of Service	Front Running	Reentrancy	Time Manipulation	Unchecked Low Level Calls	Bulk Analysis Support	Integrability	Robustness	Ease of Setup
Conkas	✓		✓	✓		✓	✓		✓		✓		✓	✓			✓		✓	✓	✓	✓	0	2	2	3
Manticore		✓	✓	✓		✓			✓	✓	✓				✓	✓	✓			✓	✓	✓	3	3	3	3
teEther	✓	✓		✓		✓					✓			✓		✓	✓						3	2	2	2
Mythril	✓		✓	✓		✓			✓		✓	✓		✓	✓	✓	✓		✓	✓		✓	0	2	3	3
Slither	✓				✓		✓		✓	✓			✓	✓		✓		✓		✓	✓	✓	0	3	3	3

Table 8. Requirement Specification for Selected Tools

The specifications are split into two groups:

1. *Objective Criteria*: The applicable options are deduced from past research work [48],

[51]–[53], [55], [56] done on the tools by academic researchers, and documentation provided by the tool creators.

2. *Subjective Criteria*: Each requirement in this category is assigned a score based on the evaluation framework proposed in Chapter 4. The scores range from 0 to 3, with 0 indicating that the feature is not applicable to the tool in question, 1 indicating that the tool provides minimal support for a requirement, 2 indicating that the tool provides good support, and 3 indicating that the tool provides excellent support for a requirement. It is critical to emphasise that the subjective criteria group scores are based on personal experience working with the tools and also on a review of the tool creator's documentation.

Secondly, for each of the constructed scenarios in the subsequent sections, weights are assigned to each of the specification requirement based on the needs of the vulnerability detection job and inputted into the Weights tab of the prototype. For this PoC prototype we have have set a range of 1-10 for the weights, where 1 means feature is less important to the vulnerability detection job and 10 means the feature is extremely important to the job. The features that apply to the tool being evaluated are entered into the evaluate tab in accordance with Table 8. Finally, the submit button is clicked, and the result output containing the utility score for each tool is recorded.

6.3.1 Scenario 1

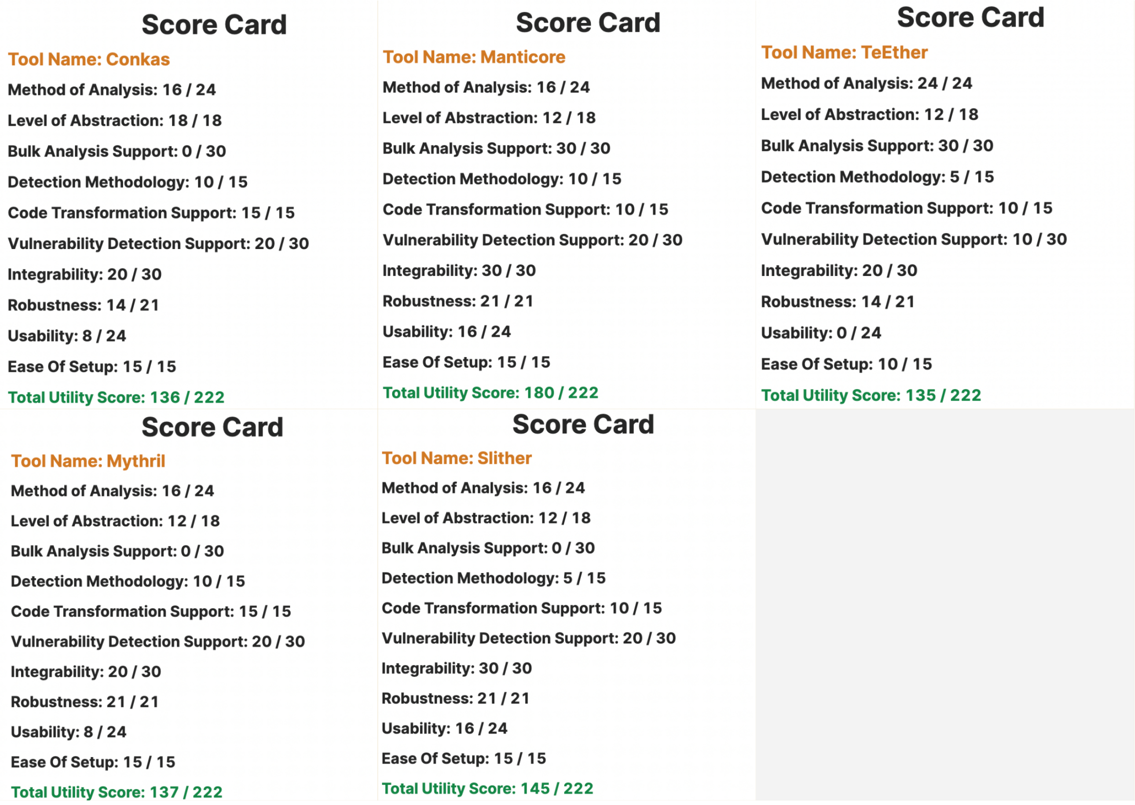
Job Description: *A newly launched decentralised cryptocurrency exchange is looking for a smart contract vulnerability detection tool that can be easily integrated into their continuous integration/continuous delivery process to identify vulnerabilities during development. Additionally, the selected tool should be capable of simultaneously scanning a large number of smart contracts.*

As implied by the job description, we can presume the most critical features relevant to the job at hand are **integrability, vulnerability detection**, and **ease of use**. Table 9 illustrates how the requirement specification is weighted according to the job description, and also shows the respective scores assigned based on the scoring guideline in Table 4.

Requirements	Vulnerability Detection Tool					
	Conkas	Manticore	teEther	Mythril	Slither	Weight
Method of Analysis	2	2	3	2	2	8
Level of Abstraction	3	2	2	2	2	6
Bulk Analysis Support	0	3	3	0	0	10
Detection Methodology	2	2	1	2	1	5
Code Transformation Support	3	2	2	3	2	5
Vulnerability Detection Support	2	2	1	2	2	10
Integrability	2	3	2	2	3	10
Robustness	2	3	2	3	3	7
Usability	1	2	0	1	2	8
Ease of Setup	3	3	2	3	3	5
Utility Score	**136**	**188**	**135**	**137**	**145**	

Table 9. Scenario 1 Utility Scores

Manticore scores demonstrate a strong focus on all of the requirements, and as a result, we can conclude that Manticore provides more utility for the job described. Figure 10 depicts the prototype's utility score output for each tool.



Score Card

Tool Name: Conkas

Method of Analysis: 16 / 24

Level of Abstraction: 18 / 18

Bulk Analysis Support: 0 / 30

Detection Methodology: 10 / 15

Code Transformation Support: 15 / 15

Vulnerability Detection Support: 20 / 30

Integrability: 20 / 30

Robustness: 14 / 21

Usability: 8 / 24

Ease Of Setup: 15 / 15

Total Utility Score: 136 / 222

Score Card

Tool Name: Manticore

Method of Analysis: 16 / 24

Level of Abstraction: 12 / 18

Bulk Analysis Support: 30 / 30

Detection Methodology: 10 / 15

Code Transformation Support: 10 / 15

Vulnerability Detection Support: 20 / 30

Integrability: 30 / 30

Robustness: 21 / 21

Usability: 16 / 24

Ease Of Setup: 15 / 15

Total Utility Score: 180 / 222

Score Card

Tool Name: TeEther

Method of Analysis: 24 / 24

Level of Abstraction: 12 / 18

Bulk Analysis Support: 30 / 30

Detection Methodology: 5 / 15

Code Transformation Support: 10 / 15

Vulnerability Detection Support: 10 / 30

Integrability: 20 / 30

Robustness: 14 / 21

Usability: 0 / 24

Ease Of Setup: 10 / 15

Total Utility Score: 135 / 222

Score Card

Tool Name: Mythril

Method of Analysis: 16 / 24

Level of Abstraction: 12 / 18

Bulk Analysis Support: 0 / 30

Detection Methodology: 10 / 15

Code Transformation Support: 15 / 15

Vulnerability Detection Support: 20 / 30

Integrability: 20 / 30

Robustness: 21 / 21

Usability: 8 / 24

Ease Of Setup: 15 / 15

Total Utility Score: 137 / 222

Score Card

Tool Name: Slither

Method of Analysis: 16 / 24

Level of Abstraction: 12 / 18

Bulk Analysis Support: 0 / 30

Detection Methodology: 5 / 15

Code Transformation Support: 10 / 15

Vulnerability Detection Support: 20 / 30

Integrability: 30 / 30

Robustness: 21 / 21

Usability: 16 / 24

Ease Of Setup: 15 / 15

Total Utility Score: 145 / 222

Figure 10. Prototype Utility Score Card

6.3.2 Scenario 2

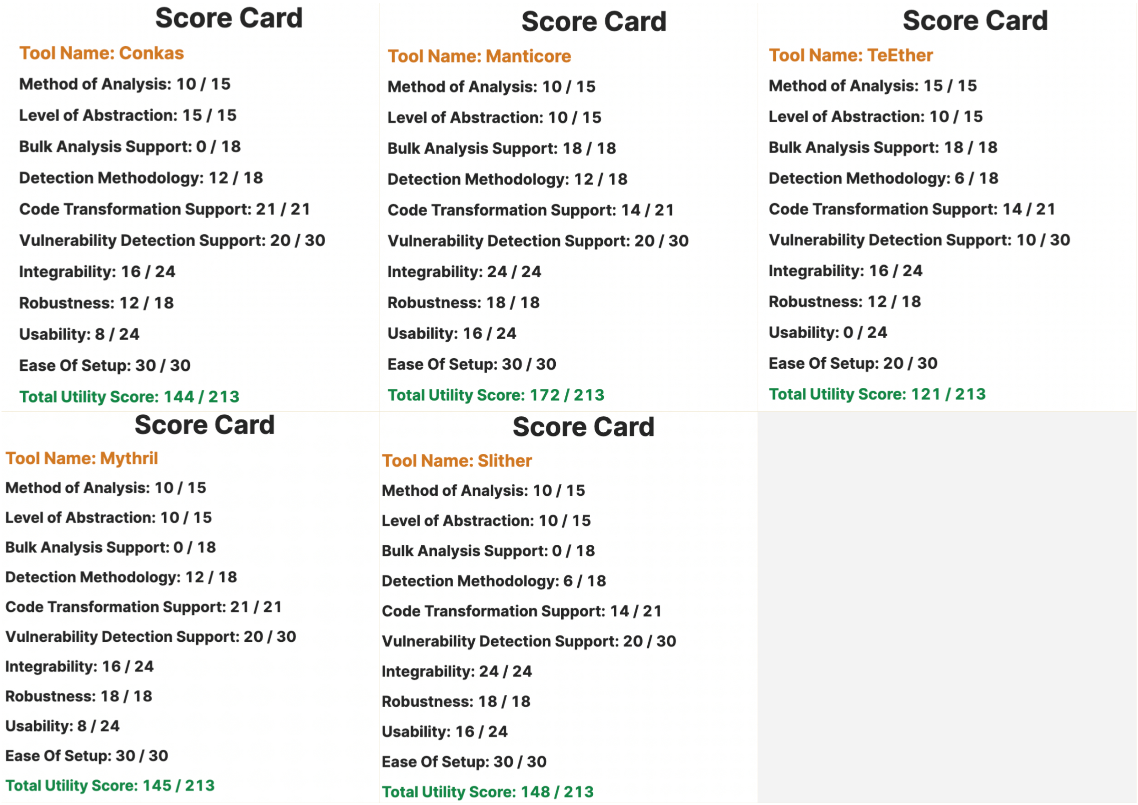
Job Description: *A web3 bug bounty platform that is developing a smart contract security training course for solidity developers is looking for a smart contract vulnerability detection tool to use as a benchmark for testing smart contracts. It is expected that developers will be required to install and use the selected tool to test a number of vulnerable smart contracts during the course.*

From the Job description, we can presume that **ease of use** and **vulnerability detection** are the most important features relevant to the job at hand, Table 10 shows how various weights are assigned to the requirement specification based on the job description.

Requirements	**Vulnerability Detection Tool**					
	Conkas	**Manticore**	**teEther**	**Mythril**	**Slither**	**Weight**
Method of Analysis	2	2	3	2	2	5
Level of Abstraction	3	2	2	2	2	5
Bulk Analysis Support	0	3	3	0	0	6
Detection Methodology	2	2	1	2	1	6
Code Transformation Support	3	2	2	3	2	7
Vulnerability Detection Support	2	2	1	2	2	10
Integrability	2	3	2	2	3	8
Robustness	2	3	2	3	3	6
Usability	1	2	0	1	2	8
Ease of Setup	3	3	2	3	3	10
Utility Score	**144**	**172**	**121**	**145**	**148**	

Table 10. Scenario 2 Utility Scores

For *scenario 2*, Manticore once again demonstrates a strong focus on requirements, owing to its ease of setup, integrability, and vulnerability detection capabilities. Additionally, Manticore's support for bulk analysis of smart contracts puts it ahead of other tools with limited bulk analysis support. Based on the evaluation results, we can conclude that Manticore is more useful for the job at hand. Figure 11 depicts the utility score output from the prototype.



Score Card

Tool Name: Conkas

Method of Analysis: 10 / 15

Level of Abstraction: 15 / 15

Bulk Analysis Support: 0 / 18

Detection Methodology: 12 / 18

Code Transformation Support: 21 / 21

Vulnerability Detection Support: 20 / 30

Integrability: 16 / 24

Robustness: 12 / 18

Usability: 8 / 24

Ease Of Setup: 30 / 30

Total Utility Score: 144 / 213

Score Card

Tool Name: Manticore

Method of Analysis: 10 / 15

Level of Abstraction: 10 / 15

Bulk Analysis Support: 18 / 18

Detection Methodology: 12 / 18

Code Transformation Support: 14 / 21

Vulnerability Detection Support: 20 / 30

Integrability: 24 / 24

Robustness: 18 / 18

Usability: 16 / 24

Ease Of Setup: 30 / 30

Total Utility Score: 172 / 213

Score Card

Tool Name: TeEther

Method of Analysis: 15 / 15

Level of Abstraction: 10 / 15

Bulk Analysis Support: 18 / 18

Detection Methodology: 6 / 18

Code Transformation Support: 14 / 21

Vulnerability Detection Support: 10 / 30

Integrability: 16 / 24

Robustness: 12 / 18

Usability: 0 / 24

Ease Of Setup: 20 / 30

Total Utility Score: 121 / 213

Score Card

Tool Name: Mythril

Method of Analysis: 10 / 15

Level of Abstraction: 10 / 15

Bulk Analysis Support: 0 / 18

Detection Methodology: 12 / 18

Code Transformation Support: 21 / 21

Vulnerability Detection Support: 20 / 30

Integrability: 16 / 24

Robustness: 18 / 18

Usability: 8 / 24

Ease Of Setup: 30 / 30

Total Utility Score: 145 / 213

Score Card

Tool Name: Slither

Method of Analysis: 10 / 15

Level of Abstraction: 10 / 15

Bulk Analysis Support: 0 / 18

Detection Methodology: 6 / 18

Code Transformation Support: 14 / 21

Vulnerability Detection Support: 20 / 30

Integrability: 24 / 24

Robustness: 18 / 18

Usability: 16 / 24

Ease Of Setup: 30 / 30

Total Utility Score: 148 / 213

Figure 11. Prototype Utility Score Card (2)

6.4 Related Work

As described in Chapter 1, a number of smart contract vulnerability detection tools are available for use. Liu and Liu [58] explore the various smart contract verification methods and highlight a number of smart contract vulnerability detection tools, but do not directly compare the tools. In addition to examining the security and performance of smart contracts, Rouhani and Deters [59] compare the performance of nine different vulnerability detection tools. However, the comparison does not take the features of the tools into account, relying instead on performance analysis. A systematic mapping and taxonomy of smart contract vulnerabilities is proposed by Zeli Wang et al [60]. Details of some vulnerability detection tools are provided but not compared. In the research conducted by Kim and Ryu [61], 27 vulnerability detection tools are evaluated in terms of their ability to detect known vulnerabilities from a specified dataset. However, the evaluation is solely based on vulnerability detection and does not account for usability or vulnerability detection use case. In a separate study by Vacca et al. [62], the characteristics of 26 tools are described, but they are not compared.

Luu et al [24] describes various smart contract vulnerabilities on the Ethereum blockchain, including code snippets and examples, but makes no use of any taxonomy or classification.

Atzei et al [32] created one of the first Ethereum vulnerability taxonomies, which also categorizes vulnerabilities into three classes. However, the proposed taxonomy is somewhat out of date, as issues such as the Stack size limit have since been resolved on the EVM.

As stated previously, numerous studies have been conducted on smart contract security tools and vulnerability classification taxonomies. However, to the author's knowledge, the work proposed in this thesis is the first time a utility evaluation framework which can be applied to the available tools to determine their utility has been developed.

6.5 Discussion

In this section, we evaluated the proposed framework using the descriptive evaluation method used in design science research. One method for evaluating the artefact of the thesis using the descriptive evaluation method is to create scenarios based on the artefact. To properly evaluate the artefact, we created a prototype in JavaScript and a web application using Retool, we chose retool for its simplicity and ease of use. On the UI, we included three tabs: *Weights tab* for assigning weight to each requirement, *Evaluate tab* for selecting features applicable to each tool, and *Scoring guideline tab* for referencing a scoring guideline to help users understand the score assigned to each option. Additionally, two scenarios are constructed around the artefact, *scenario 1* is a vulnerability detection job that requires a tool that prioritises integrability, vulnerability detection, and ease of use, and *scenario 2* is a vulnerability detection job that prioritises ease of use and vulnerability detection. While we have assigned weights to each requirement, it is critical to remember that the weights are subjective, and a user may choose to assign different weights based on the implied importance of each requirement specification to the job at hand. Finally, the tool's weights and options are entered into the prototype for utility evaluation.

6.6 Conclusion

The purpose of this chapter was to evaluate the framework and prototype using the *descriptive evaluation method* in design science research. To this end, we constructed two scenarios in which the framework would be used to evaluate the utility of a smart contract vulnerability detection tool. We created a prototype using retool and evaluated the utility of five smart contract vulnerability detection tools. The evaluation results are displayed in Table 9 and Table 10 Based on the evaluation of the prototype, with weighting assigned to each functional and non-functional requirement specification, it can be concluded that the framework provides a fair and accurate method for evaluating the utility of a smart contract vulnerability detection tool. However, it is important to note that the framework is subject to the same limitations as any evaluation framework, the weights assigned to each requirement specification is subjective, and it is the responsibility of the user to assign weights that are appropriate to their situation.

7. Conclusion and Future Work

The following chapter concludes the research done in this thesis and answers the research questions in Chapter 1, Section 7.1 provides a general conclusion of the thesis, Section 7.2 presents answers to the research questions, furthermore Section 7.3 provides limitations of the thesis and finally, future work is discussed in Section 7.4.

7.1 Conclusion

This thesis proposed a novel evaluation framework for determining the utility and suitability of a vulnerability detection tool. The framework is intended to determine a tool's utility by examining its features as functional and non-functional requirements. Additionally, the framework was designed to be adaptable, as it enables users to customise the evaluation options.

To aid in the framework's development, we examined the various available smart contract vulnerability taxonomies and chose the DASP Top 10 taxonomy due to its high-level overview of Ethereum vulnerabilities and also its similarity to the OWASP Top 10. We analysed the DASP Top 10 vulnerabilities by providing real-world attack scenarios and also discussed remediation techniques.

Following that, we presented a process for evaluating the utility of smart contract vulnerability detection tools by analysing the requirement specification for an ideal smart contract vulnerability detection tool. The requirement specifications are divided into functional and non-functional requirements and then discussed in detail. A utility evaluation equation is then presented and a scoring guideline for each requirement is provided.

After establishing the evaluation framework, we presented an overview of existing smart contract vulnerability detection tools on the Ethereum blockchain, discussed their capabilities, and then compared the tools based on categories such as analysis method, detection methodology, level of abstraction, usability, code transformation method, and vulnerability type detected. Then, using JavaScript and Retool, we created a proof-of-concept prototype and evaluated the chosen tools using the framework's prototype.

7.2 Answering The Research Questions

The main research question for this thesis is: **How can Ethereum Smart Contract Vulnerability Detection tools be evaluated?** As stated in Chapter 1, we divided the main question into three sub-questions, the following subsections provides answers to them.

7.2.1 RQ1: What are the common vulnerabilities affecting smart contracts on the Ethereum blockchain?

Based on the DASP Top 10 taxonomy we discovered that **Reentrancy, Access Control, Arithmetic, Unchecked Return Values for Low Level, Denial of Service (DoS), Bad Randomness, Front Running, Time Manipulation, Short Address,** and U**nknown Unknown** vulnerabilities are the most prevalent vulnerabilities on the Ethereum blockchain. We discovered that *reentrancy* is the most frequently used attack vector against smart contracts, and we demonstrated real-world attack scenarios for each vulnerability, including the DAO hack, the proof of weak hands exploit, and the cryptopuppies hack.

Additionally, we provided solidity code examples of vulnerable smart contracts and discussed possible mitigation strategies. Additionally, we learned the critical nature of early vulnerability detection and secure coding practises in avoiding smart contract exploits.

7.2.2 RQ2: What are the evaluation requirements for vulnerability detection tools on the Ethereum blockchain?

We concluded that to efficiently evaluate the utility of a tool, the features of the tool needs to be examined individually as functional and non-functional requirements based on the vulnerability detection use-case for a particular job. The The functional features that make up the overall utility are: *Method of analysis, level of abstraction, bulk analysis support, detection methodology, code transformation support*, and *vulnerability detection support*. The non-functional requirements refer to attributes of the tool that cannot be out-rightly observed during its execution, they include: *integrability, robustness, usability,* and *ease of set up* of the tool.

Next, we proposed the utility equation shown in Equation 4.1 and also provided a scoring guideline shown in Table 4 that assigns a score to each requirement, on the scale of {**3|2|1|0**}. Where 3 means a vulnerability detection tool has an **excellent support** for a requirement, 2 means **good support**, 1 means **minimal support** and 0 means a requirement is **not applicable** for the specific vulnerability detection tool.

7.2.3 RQ3: What are the features of the existing open-source vulnerability detection tools on the Ethereum blockchain?

We discovered over 140 smart contract vulnerability detection tools available on the Ethereum blockchain. We established a selection criteria for tools to analyse in this research, which included tools published or updated from 2021 to the present, tools that accept solidity code or bytecode as input, tools that focus exclusively on vulnerability detection, and finally, tools that require only solidity source code or bytecode for analysis, the tools that met our criteria were ***Conkas, Manticore, teEther, Mythril*** and ***Slither***.

We conclude that that all the tools have a strong focus on the types of vulnerabilities that are listed in the DASP top 10 vulnerability taxonomy. we showed that the tools have a fair amount of overlap in terms of the type of vulnerabilities detected by each tool. In terms of usability, we discovered that there are many Ethereum smart contract detection tools that support a graphical user interface (GUI), command line interface (CLI), In terms of the level of abstraction, we discovered that most Ethereum smart contract detection tools support bytecode abstractions.

We also learnt that there are currently no tools that support all the DASP top 10 vulnerabilities, we can conclude that the security of Ethereum smart contract is in the early stage of development, and many tools are not mature enough to provide a comprehensive security analysis.

7.3 Limitations

The primary limitation of this thesis is that it is limited to the Ethereum blockchain, there are several other blockchain available, including EOS, Cosmos, and Avalanche. While the work in this thesis may still be applicable to EVM-based chains, the thesis's sole focus was on the Ethereum blockchain, which means that some of the vulnerabilities discussed in this thesis may not be applicable to other blockchains, secondly, the scope of this thesis is limited to smart contracts written in Solidity language, although there are several other programming languages, including Vyper, YUL+, and FE used in developing smart contracts on the Ethereum blockchain.

Although the DASP Top 10 vulnerability was chosen for this thesis, there are several other vulnerability taxonomies available, which limits our approach to a subset of all possible vulnerabilities. Additionally, our framework in its current form examines only a subset of tool features, and the framework's weight assignment process is subjective, which means that the result may vary across different users.

Concerning the tools evaluated in this study, the selection criteria could be expanded to allow for the evaluation of additional tools using the framework, additionally, the scenarios constructed to evaluate the prototype are limited to two examples, which may not cover all use case scenarios.

7.4 Future Work

There are numerous avenues for future research in this area, some of which building on the work presented in this thesis. One such future research is expanding the scope of the framework to allow for the evaluation of additional blockchain smart contract detection tools, this will allow for a comprehensive analysis of all available smart contract detection tools.

Another area of future work is the development of an improved method of weight assignment based on the requirements, which will reduce subjectivity when conducting the evaluation process. Additionally, the tools evaluated in this thesis used a subset of the DASP Top 10 vulnerabilities, which means that the tools may have a higher or lower detection rate for additional vulnerabilities. To accurately evaluate the utility of a tool, the tool needs to be evaluated using the evaluation framework prototype using a larger set of scenarios.

The framework presented in this thesis can be extended to include additional features, for instance, the framework can be extended to support the evaluation of tools that are not available on the Ethereum blockchain, also, the framework can be extended to support additional vulnerability taxonomies. The framework only evaluates the capabilities of a tool and does not provide an evaluation of the tool's accuracy, but it is possible to extend the evaluation framework to test the accuracy of the tools.

To enable the evaluation of additional smart contract vulnerability detection tools, a tool repository needs to be created that contains all open-source tools available on the Ethereum blockchain. Such a repository needs to be accessible through a web interface that can be used to query tools based on the type of vulnerability detected, tool features, and other criteria, this will enable users to easily select the most suitable tool for a specific job.

References

[1] Synopsys. “Blockchain.” (2022), [Online]. Available: https://www.synopsys.com/glossary/what-is-blockchain.html.

[2] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” in *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008.

[3] N. Reiff, *Bitcoin vs. ethereum: What’s the difference?* [Accessed: 18-03-2022]. [Online]. Available: https://www.investopedia.com/articles/investing/031416/bitcoin-vs-ethereum-driven-different-purposes.asp.

[4] M. D. Pierro, “What Is the Blockchain?” *Computing in Science and Engineering*, vol. 19, no. 5, 2017, ISSN: 15219615. DOI: 10.1109/MCSE.2017.3421554.

[5] C. Catalini, “The Potential for Blockchain to Transform Electronic Health Records,” *Harvard Business Review*, 2017.

[6] V. Buterin. “Ethereum: A next-generation smart contract and decentralized application platform.” (2015), [Online]. Available: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_White_Paper_-_Buterin_2014.pdf.

[7] S. Tual. “Ethereum launches.” (2015), [Online]. Available: https://blog.ethereum.org/2015/07/30/ethereum-launches/.

[8] C. Staff. “Why are most dapps built on ethereum?” (2021), [Online]. Available: https://www.gemini.com/cryptopedia/dapps-ethereum-decentralized-application.

[9] M. Faizan, T. Brenner, F. Foerster, C. Wittwer, and B. Koch, “Decentralized bottom-up energy trading using ethereum as a platform,” *Journal of Energy Markets*, vol. 12, no. 2, 2019, ISSN: 17563615. DOI: 10.21314/JEM.2019.193.

[10] W. Zou, D. Lo, P. S. Kochhar, *et al.*, “Smart Contract Development: Challenges and Opportunities,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, 2021, ISSN: 19393520. DOI: 10.1109/TSE.2019.2942301.

[11] N. Szabo, “Smart Contracts : Building Blocks for Digital Markets,” *EXTROPY: The Journal of Transhumanist Thought*, vol. 18, no. 2, 1996.

[12] O. G. Güçlütürk, “The DAO Hack Explained: Unfortunate Take-off of Smart Contracts,” *Medium.com*, 2018. [Online]. Available: `https://medium.com/@oguclututurk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562`.

[13] Y. Ni, C. Zhang, and T. Yin, *A Survey of Smart Contract Vulnerability Research*, 2020. DOI: `10.19363/J.cnki.cn10-1380/tn.2020.05.07`.

[14] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, “TXSPECTOR: Uncovering attacks in ethereum from transactions,” in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[15] D. Wang, B. Jiang, and W. K. Chan, “WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection,” *arXiv*, 2020, ISSN: 23318422.

[16] M. Staderini and A. Bondavalli, “Investigating Static Analyzers Detection Capabilities on Ethereum Smart Contracts,” *Proceedings of the 28th Mini-Symposium of the Department of Measurement and Information Systems, Budapest University of Technology and Economics*, 2021.

[17] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, *Smart contract security: A software lifecycle perspective*, 2019. DOI: `10.1109/ACCESS.2019.2946988`.

[18] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, “Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models,” *IEEE Access*, vol. 8, 2020, ISSN: 21693536. DOI: `10.1109/ACCESS.2020.2969429`.

[19] A. López Vivar, A. L. Sandoval Orozco, and L. J. García Villalba, “A security framework for Ethereum smart contracts,” *Computer Communications*, vol. 172, 2021, ISSN: 1873703X. DOI: `10.1016/j.comcom.2021.03.008`.

[20] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Quarterly: Management Information Systems*, vol. 28, no. 1, 2004, ISSN: 02767783. DOI: `10.2307/25148625`.

[21] E. Sunday, *Top 5 smart contract programming languages for blockchain?* [Accessed: 18-03-2022]. [Online]. Available: `https://blog.logrocket.com/smart-contract-programming-languages/`.

[22] N. Mehrotra, *An introduction to solidity, the language that runs ethereum*, [Accessed: 18-03-2022]. [Online]. Available: `https://www.opensourceforu.com/2019/08/an-introduction-to-solidity-the-language-that-runs-ethereum/`.

[23] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *POST*, 2017.

[24] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 254–269, ISBN: 9781450341394. DOI: 10.1145/2976749.2978309. [Online]. Available: https://doi.org/10.1145/2976749.2978309.

[25] D. Perez and B. Livshits, “Smart contract vulnerabilities: Vulnerable does not imply exploited,” in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1325–1341, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/perez.

[26] Coinbase, *What is a dex?* [Accessed: 18-03-2022]. [Online]. Available: https://www.coinbase.com/learn/crypto-basics/what-is-a-dex.

[27] G. A. Pierro, R. Tonelli, and M. Marchesi, “An organized repository of ethereum smart contracts’ source codes and metrics,” *Future Internet*, vol. 12, p. 197, Nov. 2020. DOI: 10.3390/fi12110197.

[28] J. Baltrusaitis, *Total value locked in defi surges over 1,200% in 2021 to surpass $240 billion*, [Accessed: 20-03-2022]. [Online]. Available: https://finbold.com/total-value-locked-in-defi-surges-over-1200-in-2021-to-surpass-240-billion.

[29] G. Keyes, “Defi tops $100 billion for first time as cryptocurrencies surge",” en, in *Bloomberg. Archived from the original on 2021-11-04*, Retrieved 2022-03-20., Oct. 20, 2021.

[30] Elliptic, *Defi: Risk, regulation, and the rise of decrime*, [Accessed: 20-03-2022]. [Online]. Available: https://www.elliptic.co/resources/defi-risk-regulation-and-the-rise-of-decrime.

[31] S. Dhyan Raj, *What financial institutions can learn from the ethereum parity hack*, [Accessed: 20-03-2022]. [Online]. Available: https://www.synechron.com/sites/default/files/white-paper/what-financial-institutions-can-learn-from-the-thereum-parityhack.pdf.

[32] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186, ISBN: 978-3-662-54455-6.

[33] Z. A. Khan and A. S. Namin, *A survey on vulnerabilities of ethereum smart contracts*, 2020. DOI: 10.48550/ARXIV.2012.14481. [Online]. Available: https://arxiv.org/abs/2012.14481.

[34] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[35] N. Group *et al.*, *Decentralized application security project (dasp) top 10*, 2018.

[36] P. Technologies, *A postmortem on the parity multi-sig library self-destruct.* [Online]. Available: `https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.

[37] R. Browne, *’accidental’ bug may have frozen* $280millionworthofdigitalcoinetherinacryptocurren$ Nov. 2017. [Online]. Available: `https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html`.

[38] A. P. C. Monteiro, “A study of static analysis tools for ethereum smart contracts,” 2019.

[39] D. A. Manning, *Solidity security: Comprehensive list of known attack vectors and common anti-patterns*, Oct. 2018. [Online]. Available: `https://blog.sigmaprime.io/solidity-security.html#ouflow`.

[40] NVD, *Cve-2018-10299 detail*, Apr. 2018. [Online]. Available: `https://nvd.nist.gov/vuln/detail/CVE-2018-10299`.

[41] KotET, *Post-mortem investigation (feb 2016)*, Feb. 2016. [Online]. Available: `https://www.kingoftheether.com/postmortem.html`.

[42] Y. Smaragdakis, *Bad randomness is even dicier than you think*, Mar. 2019. [Online]. Available: `https://media.dedaub.com/bad-randomness-is-even-dicier-than-you-think-7fa2c6e0c2cd`.

[43] B. Powers, *New research sheds light on the front-running bots in ethereum’s dark forest*, Dec. 2020. [Online]. Available: `https://www.coindesk.com/tech/2020/12/29/new-research-sheds-light-on-the-front-running-bots-in-ethereums-dark-forest/`.

[44] N. F. Samreen and M. H. Alalfi, “A survey of security vulnerabilities in ethereum smart contracts,” 2021. DOI: `10.48550/ARXIV.2105.06974`. [Online]. Available: `https://arxiv.org/abs/2105.06974`.

[45] H. Chen, M. Pendleton, L. Njilla, and S. Xu, “A survey on ethereum systems security: Vulnerabilities, attacks, and defenses,” *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020, ISSN: 0360-0300. DOI: `10.1145/3391195`. [Online]. Available: `https://doi.org/10.1145/3391195`.

[46] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[47] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh, “MPro: Combining static and symbolic analysis for scalable testing of smart contract,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, Oct. 2019. DOI: `10.1109/issre.2019.00052`. [Online]. Available: `https://doi.org/10.1109%2Fissre.2019.00052`.

[48] M. di Angelo and G. Salzer, “A survey of tools for analyzing ethereum smart contracts,” in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019, pp. 69–78. DOI: `10.1109/DAPPCON.2019.00018`.

[49] F. E. Allen, “Control flow analysis,” in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19, ISBN: 9781450373869. DOI: `10.1145/800028.808479`. [Online]. Available: `https://doi.org/10.1145/800028.808479`.

[50] A. Moona and R. Mathew, “Review of tools for analyzing security vulnerabilities in ethereum based smart contracts,” Jan. 2021. DOI: `10.2139/ssrn.3769774`.

[51] H. Rameder, M. di Angelo, and G. Salzer, “Review of automated vulnerability analysis of smart contracts on ethereum,” *Frontiers in Blockchain*, vol. 5, 2022, ISSN: 2624-7852. DOI: `10.3389/fbloc.2022.814977`. [Online]. Available: `https://www.frontiersin.org/article/10.3389/fbloc.2022.814977`.

[52] N. Veloso, “Conkas: A modular and static analysis tool for ethereum bytecode,” 2021.

[53] J. Krupp and C. Rossow, “teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts,” in *27th USENIX Security Symposium (USENIX Security 18)*, USENIX Association, 2018. [Online]. Available: `https://publications.cispa.saarland/2612/`.

[54] B. Mueller, “Smashing ethereum smart contracts for fun and real profit,” *HITB SECCONF Amsterdam*, vol. 9, p. 54, 2018.

[55] R. Fontein, “Comparison of static analysis tooling for smart contracts on the evm,” in *28th Twente Student conference on IT*, 2018.

[56] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008.

[57] Retool, *Build internal tools, remarkably fast.* [Online]. Available: https://retool.com/.

[58] J. Liu and Z. Liu, “A survey on security verification of blockchain smart contracts,” *IEEE Access*, vol. 7, pp. 77 894–77 904, 2019. DOI: 10.1109/ACCESS.2019.2921624.

[59] S. Rouhani and R. Deters, “Security, performance, and applications of smart contracts: A systematic survey,” *IEEE Access*, vol. 7, pp. 50 759–50 779, 2019.

[60] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, “Ethereum smart contract security research: Survey and future research opportunities,” *Frontiers of Computer Science*, vol. 15, no. 2, pp. 1–18, 2021.

[61] S. Kim and S. Ryu, “Analysis of blockchain smart contracts: Techniques and insights,” in *2020 IEEE Secure Development (SecDev)*, IEEE, 2020, pp. 65–73.

[62] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, “A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges,” *Journal of Systems and Software*, vol. 174, p. 110 891, 2021.

Appendix 1 – Non-exclusive license for reproduction and publication of a graduation thesis[1]

I David, Isaac Mayowa

1. Grant Tallinn University of Technology free license (non-exclusive license) for my thesis "An Evaluation Framework for Smart Contract Vulnerability Detection Tools on the Ethereum Blockchain", supervised by Alexander Norta and Toomas Lepikult
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive license.
3. I confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

May 16, 2022

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Utility.js

```javascript

const evaluation = {

  methodOfAnalysis: {
    1: 2, // number of options selected : score for each option
    2: 3
  },
  levelOfAbstraction: {
    1: 2,
    2: 3,

  },
  bulkAnalysisSupport: {
    "Excellent Support (3 Points)": 3,
    "Good Support (2 Points)": 2,
    "Minimal Support (1 Point)": 1,
    "Not Applicable (0 Point)": 0
  },
  detectionMethodology: {
    1: 1,
    2: 2,
    3: 3
  },
  codeTransformationSupport: {
    1: 1,
    2: 2,
    3: 3
  },
  vulnerabilityDetectionSupport: {
    0: 0,
    2: 1,
    4: 2,
    7: 3
  },
  integrability: {
    "Excellent Support (3 Points)": 3,
    "Good Support (2 Points)": 2,
    "Minimal Support (1 Point)": 1,
    "Not Applicable (0 Point)": 0
  },
  robustness: {
    "Excellent Support (3 Points)": 3,
    "Good Support (2 Points)": 2,
    "Minimal Support (1 Point)": 1,
    "Not Applicable (0 Point)": 0
  },
  usability: {
    1: 1,
    2: 2,
    3: 3
  },
  easeOfSetup: {
    "Excellent Support (3 Points)": 3,
    "Good Support (2 Points)": 2,
    "Minimal Support (1 Point)": 1
  }
}

function getScorePerUtility(groupName) {
  let score = 0;
  let valueLength = groupName.value.length

  switch (true) {
    case vulnerabilityDetectionSupport === groupName:
      let evalDict = evaluation[groupName.formDataKey]

      if ([0, 1].includes(valueLength)) {
        score = evalDict[0]
      } else if ([2, 3].includes(valueLength)) {
        score = evalDict[2]
      } else if ([4, 5, 6].includes(valueLength)) {
        score = evalDict[4]
      } else {
        score = evalDict[7]
      }
      break;

    case [bulkAnalysisSupport, integrability, robustness, easeOfSetup].
        includes(groupName):
      score = evaluation[groupName.formDataKey][groupName.value[0]]
      break;

    default:
      let length = valueLength >= 3 ? 3 : valueLength
      score = evaluation[groupName.formDataKey][length]
  }

  return score

}

function calculateScore(){
  let weights = Object.keys(form1.data)
  let utilities = [methodOfAnalysis, levelOfAbstraction, bulkAnalysisSupport, detectionMethodology, codeTransformationSupport, vulnerabilityDetectionSupport, integrability, robustness, usability, easeOfSetup]

  let results = utilities.map((k, i) => (getScorePerUtility(k) || 0) * form1.data[weights[i]])

  // Sum up results and return total
  // return results.reduce((a, b) => a + b, 0)
  return results

}

return calculateScore()
```