Vulnerability Detection in Blockchain Smart Contracts Using Cloud-Based Automated Machine Learning

by Arthiraiyan Appulingam

B.S. in Computer Engineering, May 2002, Texas A&M University
Master of Business Administration, May 2006, University of Houston-Victoria
M.S. in Information Systems Engineering, May 2020, Johns Hopkins University

A Praxis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Engineering

August 31, 2023

Praxis directed by

John Sahlin
Professorial Lecturer of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Arthiraiyan Appulingam has passed the Final Examination for the degree of Doctor of Engineering as of August 2, 2023. This is the final and approved form of the Praxis.

Vulnerability Detection in Blockchain Smart Contracts Using Cloud-Based Automated Machine Learning

Arthiraiyan Appulingam

Praxis Research Committee:

John Sahlin, Professorial Lecturer of Engineering and Applied Science, Praxis Director

Shahryar Sarkani, Adjunct Professor of Engineering Management and Systems Engineering, Committee Member

John Fossaceca, Professorial Lecturer of Engineering and Applied Science, Committee Member

© Copyright 2023 by Arthiraiyan Appulingam
All rights reserved

Dedication

This praxis is dedicated to my late father, who instilled in me a passion for learning and the pursuit of knowledge.

Acknowledgments

I wish to thank my research advisor, Dr. John Sahlin, and the George Washington University School of Engineering and Applied Science faculty for their instruction, guidance, and support. I am also grateful to my classmates for their friendship and camaraderie. Special thanks to Microsoft for sponsoring access to their cloud platform for the experimental part of this praxis. And lastly, I would like to express gratitude to my family for their unwavering support and encouragement during this journey, despite the loss of countless evenings and weekends together during these past years.

Abstract of Praxis

Vulnerability Detection in Blockchain Smart Contracts Using Cloud-Based Automated Machine Learning

As software code, blockchain smart contracts often have vulnerabilities that are susceptible to cyberattacks. The use of smart contracts to store and transfer billions of dollars in value, coupled with the increasing number of known vulnerabilities, makes them an attractive target for hackers. Rapidly and accurately detecting vulnerabilities in smart contracts would assist developers with addressing the flaws and prevent vigilant parties from participating in transactions associated with vulnerable smart contracts.

Prior research focuses on the use of static and dynamic code analysis tools, and traditional machine learning built models to detect smart contract vulnerabilities. These approaches to vulnerability detection, however, present several challenges and limitations. This praxis examines the use of cloud-based automated machine learning to build smart contract vulnerability detection models using native smart contract source code, without an intermediary representation, to detect insecure arithmetic, block values as a proxy for time, reentrancy, locked Ether, and transaction order dependency flaws. The built binary classification-based vulnerability detection models were then evaluated for effectiveness through statistical performance metrics such as accuracy and precision. The research concludes that cloud-based automated machine learning and native smart contract source code can be used to build models that detect several types of vulnerabilities with a high degree of accuracy and precision. The research results also demonstrated that cloud-based automated machine learning can be used to build effective vulnerability detection models in a matter of hours, and the built models can evaluate a

smart contract for vulnerabilities within a fraction of a second - a substantial improvement over traditional static code analysis and dynamic code analysis tools, and comparable to traditional machine learning based detection models.

Table of Contents

Dedication ..... **iv**

Acknowledgments ..... **v**

Abstract of Praxis ..... **vi**

List of Figures ..... **xii**

List of Tables ..... **xiv**

List of Equations ..... **xv**

List of Acronyms ..... **xvi**

Glossary of Terms ..... **xvii**

Chapter 1—Introduction 1

1.1 Background 1

1.2 Research Motivation 3

1.3 Problem Statement 5

1.4 Thesis Statement 6

1.5 Research Objectives 6

1.6 Research Questions and Hypotheses 7

1.6.1 Research Questions 7

1.6.2 Hypotheses 7

1.7 Scope of Research 8

1.8 Research Limitations 9

1.9 Organization of Praxis 10

Chapter 2—Literature Review 11

2.1 Introduction 11

2.2 Blockchain Smart Contracts 11
2.3 Smart Contract Vulnerabilities 14
2.3.1 Insecure Arithmetic 15
2.3.2 Block Values as a Proxy for Time 16
2.3.3 Reentrancy 16
2.3.4 Locked Ether 17
2.3.5 Transaction Order Dependency 17
2.4 Machine Learning 18
2.5 Cloud-Based Automated Machine Learning 20
2.5.1 Data Preprocessing and Feature Engineering 21
2.5.2 Model Selection and Hyperparameter Optimization 22
2.5.3 Model Evaluation 23
2.6 Smart Contract Vulnerability Detection 23
2.6.1 Detection Using Static Code Analysis Tools 23
2.6.2 Detection Using Dynamic Code Analysis Tools 25
2.6.3 Detection Using Machine Learning 25
2.7 Summary and Conclusion 27
Chapter 3—Methodology 30
3.1 Introduction 30
3.2 Data Collection and Analysis 31
3.2.1 Data Sources 31
3.2.2 Data Collection 32
3.2.3 Exploratory Data Analysis 33

3.2.4 Data Preparation 33
3.3 Machine Learning Environment 35
3.4 Model Selection and Training 37
3.5 Model Evaluation 38
3.5.1 Confusion Matrix 39
3.5.2 Accuracy 40
3.5.3 Precision 40
3.5.4 Recall 41
3.5.5 F1 Score 41
3.5.6 Matthews Correlation Coefficient 42
3.5.7 Duration 42
Chapter 4—Results 43
4.1 Introduction 43
4.2 Exploratory Data Analysis 43
4.3 Data Preparation 48
4.4 Hypothesis 1 Results 52
4.5 Hypothesis 2 Results 54
4.6 Hypothesis 3 Results 57
4.7 Hypothesis 4 Results 60
4.8 Hypothesis 5 Results 63
4.9 Hypothesis 6 Results 66
4.10 Results Summary 69
Chapter 5—Discussion and Conclusions 75

5.1 Discussion 75

5.1.1 Verification of Hypotheses 75

5.1.2 Response to Research Questions 79

5.2 Conclusion 80

5.3 Contributions to Body of Knowledge 81

5.4 Recommendations for Future Research 81

References 83

Appendix A 94

List of Figures

Figure 1—Operational View of an AutoML-Based Vulnerability Detection Model 5

Figure 2—Example Solidity Source Code 13

Figure 3—Popular AutoML Platforms 20

Figure 4—Overview of Primary AutoML Steps 21

Figure 5—Research Methodology Overview 30

Figure 6—Training and Testing Dataset Preparation Sets 35

Figure 7—Confusion Matrix Format 40

Figure 8—Smart Contract Dataset Summary Statistics 44

Figure 9—Word Cloud of Top 300 Words in Smart Contract Source Code Dataset 45

Figure 10—Top 30 Word Pairs in Smart Contract Source Code Dataset 46

Figure 11—Vulnerability Sets in the ScrawlD Dataset 47

Figure 12—Percentage of Vulnerable Smart Contracts in the ScrawlD Dataset 47

Figure 13—Distribution of Vulnerabilities in the ScrawlD Dataset 48

Figure 14—Distribution of Vulnerabilities in Training Datasets 51

Figure 15—Distribution of Vulnerabilities in Test Datasets 51

Figure 16—Confusion Matrix for CPU-Based General Vulnerability Detection Model . 54

Figure 17—Confusion Matrix for GPU-Based General Vulnerability Detection Model . 54

Figure 18—Confusion Matrix for CPU-Based Insecure Arithmetic Flaws Detection Model 56

Figure 19—Confusion Matrix for GPU-Based Insecure Arithmetic Flaws Detection Model 57

Figure 20—Confusion Matrix for CPU-Based Block Values as a Proxy for Time Flaws Detection Model 59
Figure 21—Confusion Matrix for GPU-Based Block Values as a Proxy for Time Flaws Detection Model 60
Figure 22—Confusion Matrix for CPU-Based Reentrancy Flaws Detection Model 62
Figure 23—Confusion Matrix for GPU-Based Reentrancy Flaws Detection Model 63
Figure 24—Confusion Matrix for CPU-Based Locked Ether Flaws Detection Model 65
Figure 25—Confusion Matrix for GPU-Based Locked Ether Flaws Detection Model 66
Figure 26—Confusion Matrix for CPU-Based Transaction Order Dependency Flaws Detection Model 68
Figure 27—Confusion Matrix for GPU-Based Transaction Order Dependency Flaws Detection Model 69
Figure 28—Overall Test Results for CPU-Based Vulnerability Detection Models 72
Figure 29—Overall Test Results for GPU-Based Vulnerability Detection Models 72
Figure 30—Model Training Time 74
Figure 31—Average per Smart Contract Testing Time 74

List of Tables

Table 1—Common Ethereum Smart Contract Vulnerabilities 2

Table 2—Organization of Experiments 38

Table 3—Training Datasets Overview 49

Table 4—Test Datasets Overview 50

Table 5—General Vulnerability Detection Model Test Results 53

Table 6—Insecure Arithmetic Flaws Detection Model Test Results 56

Table 7—Block Values as a Proxy for Time Flaws Detection Model Test Results 59

Table 8—Reentrancy Flaws Detection Model Test Results 62

Table 9—Locked Ether Flaws Detection Model Test Results 65

Table 10—Transaction Order Dependency Flaws Detection Model Test Results 68

Table 11—Overall Test Results for CPU-Based Vulnerability Detection Models 70

Table 12—Overall Test Results for GPU-Based Vulnerability Detection Models 71

Table 13—Training and Testing Duration for GPU and CPU-Based Models 73

Table 14—Verification of Hypotheses 78

Table 15—Mapping of ScrawlD Dataset Vulnerability Labels to Prepared Datasets 94

List of Equations

Equation 1—Formula for Accuracy 40

Equation 2—Formula for Precision 41

Equation 3—Formula for Recall 41

Equation 4—Formula for F1 Score 41

Equation 5—Formula for Matthews Correlation Coefficient 42

List of Acronyms

API	Application Programming Interface
AutoML	Automated Machine Learning
CPU	Central Processing Unit
EDA	Exploratory Data Analysis
EVM	Ethereum Virtual Machine
FN	False Negative
FP	False Positive
GB	Gigabyte
GPU	Graphics Processing Unit
LightGBM	Light Gradient-Boosting Machine
MCC	Matthews Correlation Coefficient
RAM	Random Access Memory
TN	True Negative
TP	True Positive
US	United States
XGBoost	Extreme Gradient Boosting

Glossary of Terms

Blockchain	Distributed digital ledger of cryptographically signed transactions that are grouped into blocks (NIST, 2023).
Cloud Computing	Computing model that enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources in a manner that can be rapidly provisioned and released with minimal service provider interaction (NIST, 2023).
Dynamic Code Analysis Tool	A tool that analyzes computer software by executing programs built from the software being analyzed on a real or virtual processor and observing its behavior, probing the application and analyzing application responses (NIST, 2023).
Machine Learning	Machine learning is a branch of artificial intelligence which focuses on the use of algorithms and data to mimic the way that humans learn (IBM, 2023).
Smart Contracts	A collection of code and data (sometimes referred to as functions and state) that is deployed using

	cryptographically signed transactions on the blockchain network (NIST, 2023).
Static Code Analysis Tool	A tool that analyzes source code without executing the code. Static code analyzers are designed to review bodies of source code (at the programming language level) or compiled code (at the machine language level) to identify poor coding practices. Static code analyzers provide feedback to developers during the code development phase on security flaws that might be introduced into code. (NIST, 2023)
Vulnerability	Weakness in an information system that could be exploited or triggered by a threat source (NIST, 2023).

Chapter 1—Introduction

1.1 Background

There has been tremendous growth in the adoption of blockchain technologies since the introduction of Bitcoin over a decade ago (Luu et al., 2016). A popular application of blockchain technology has been their use with smart contracts. Blockchain based smart contracts are scripts or programs embedded within a blockchain that facilitate the automatic execution of transactions when set conditions are met. Since the conditions or rules associated with the transaction are programmed into the smart contract logic, there is no need for third-party intermediaries. The elimination of third-party intermediaries allows for faster transaction execution and lower costs. In addition, given the distributed nature of blockchain technology, smart contracts are immutable and transparent, thereby increasing transactional integrity and reducing the overall risk of fraud. These benefits make smart contracts the backbone of many decentralized financial transactions, including payments, lending, borrowing, insurance, and trading services. Ethereum, a popular blockchain platform, hosts millions of smart contracts transacting billions of US dollars in assets (Miller et al., 2018). During Q1 2022 alone, over 1.45 million smart contracts were created on the Ethereum platform (Alchemy, 2022).

As with other software technologies, however, smart contracts can be prone to software flaws and vulnerabilities, exposing them to cyberattacks. Furthermore, factors such as quick development cycles and the emerging nature of the technologies involved place smart contracts at an increased risk of a variety of security vulnerabilities. Smart contracts are thus enticing to malicious hackers and others seeking to exploit these weaknesses to undermine the integrity and availability of the transactions and stored

assets (Tikhomirov et al., 2018). On the Ethereum blockchain, for example, there have been several notable smart contract hacks in the past few years. In 2016, malicious hackers leveraged a common smart contract coding flaw to steal over $60 million from a decentralized autonomous organization (Miller et al., 2018). Then, in 2017, the Parity Wallet attack led to the loss of over $30 million (Palladino, 2017). In 2018, another hack exploited insecure arithmetic flaws in smart contracts, leading to the generation of enormous numbers of crypto tokens, causing reputational damage and an 8% drop in overall Ethereum value in a single day (Zhang, 2019). According to Defiyield's REKT database, there have been over $1.2 billion in losses through smart contract-related exploits just in the last few years (Hacken, 2022). Five common types of Ethereum smart contract vulnerabilities are depicted in Table 1.

Vulnerability Type	Description
Insecure arithmetic	Vulnerabilities in smart contract source code resulting from the improper handling of integer variables.
Block values as a proxy for time	Flaws resulting from the use of local node time or block number for functions within smart contracts.
Reentrancy	Issues arising from poor concurrency management in transactions between multiple smart contracts.
Locked Ether	Defects leading to the inaccessibility of value stored within smart contracts.
Transaction order dependency	Race conditions within smart contracts that occur when new transactions are added to the blockchain.

Table 1—Common Ethereum Smart Contract Vulnerabilities

Given the immutable nature of blockchain technologies such as smart contracts, early detection and remediation of vulnerabilities is essential to minimizing the risk of financial theft and other exploitation. While several tools and approaches are available to help reduce this risk, creating rapidly deployable and widely accessible vulnerability detection solutions remains a challenge.

1.2 Research Motivation

As the use of smart contracts increases and the assets they hold continue to grow, they present an increasingly attractive target for criminals. Malicious hackers seek out smart contracts with software flaws and weaknesses to exploit them for financial theft or other illicit purposes. Due to the open and public nature of popular blockchain smart contract platforms such as Ethereum, the source code for smart contracts is readily available, enabling malicious hackers to uncover exploitable weaknesses more readily than with other types of software. Alkhalifah et al. (2019) state in their study on blockchain cybersecurity incidents that there has been an upward trend since 2016 in the number of attacks targeting vulnerabilities in Ethereum smart contracts. As a result, it is critical for smart contract developers to identify source code vulnerabilities prior to deploying them to the Ethereum platform.

Several static and dynamic code analysis tools can assist with detecting security flaws in smart contracts. However, they require specialized expertise and ample time to achieve results. A potential alternative to static and dynamic code analysis tools is a machine learning-based solution. Machine learning is a contemporary field within artificial intelligence focused on harnessing the power of computing to build and train computer models that can be used to solve complex problems. While there has been

research into using machine learning for vulnerability detection in smart contracts, turnkey solutions are not widely available. Moreover, it is not trivial for the typical developer to build and train machine learning models without specialized skills and expertise in data science and coding (Lorenza et al., 2021). Thus, machine learning may not be readily accessible to the general smart contract community.

Automated machine learning (AutoML) handles many complex tasks associated with traditional machine learning, such as preprocessing, feature extraction, model selection, and postprocessing, without custom coding and limited human involvement (Escalante, 2020). In addition, AutoML is available as an on-demand service from several leading cloud service providers, further reducing the barrier to entry to this new technology. Cloud-based AutoML makes machine learning more accessible to those without the advanced machine learning skills and expertise or the time needed to handle tedious tasks such as feature extraction, scripting, and model selection.

A cloud-based AutoML approach can provide individuals and organizations with a means to rapidly build detection models to evaluate their smart contracts for vulnerabilities based on historical or other benchmark data before deployment to the Ethereum platform. Figure 1 depicts an operational view of an AutoML-based vulnerability detection model.

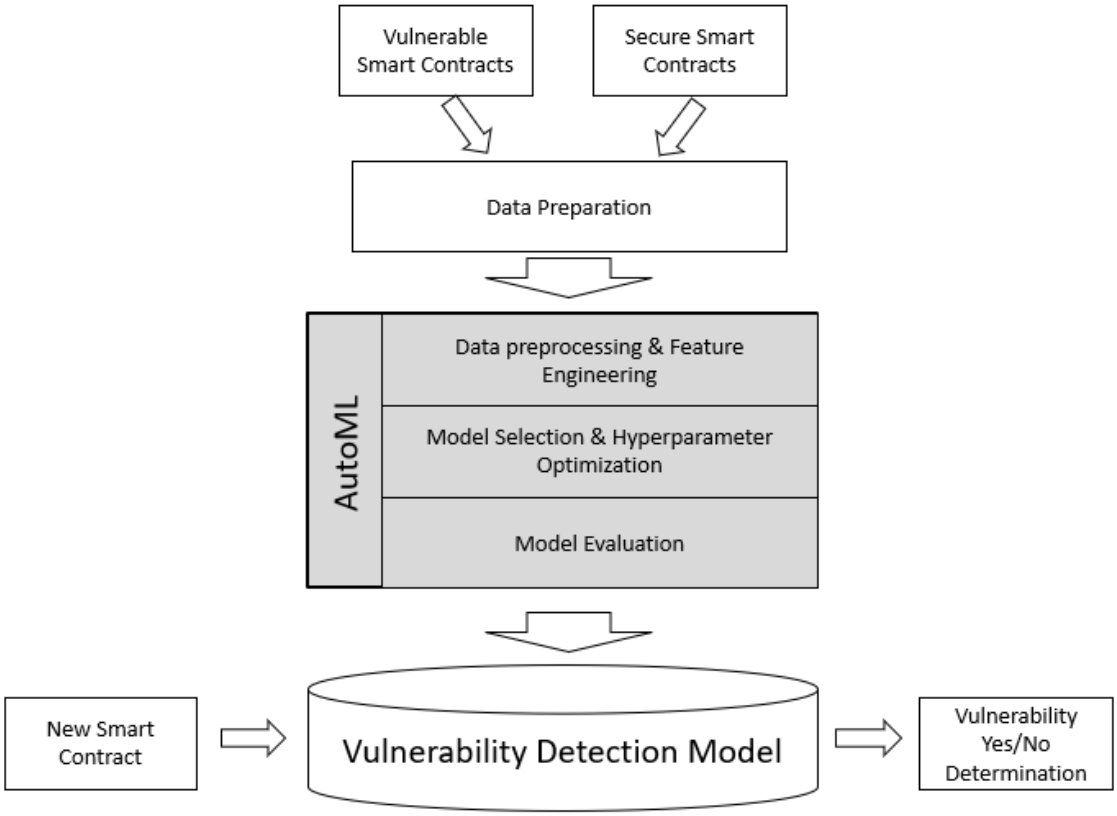

Figure 1—Operational View of an AutoML-Based Vulnerability Detection Model

1.3 Problem Statement

The ever-increasing number of defects/vulnerabilities with smart contracts, coupled with their handling of billions of dollars, make them an attractive target for hackers (Sürücü et al., 2022).

Smart contracts with vulnerabilities pose a significant risk to the assets they hold. There are several known instances of hackers exploiting smart contract vulnerabilities to steal millions of dollars. The risk of vulnerability exploitation continues to trend upward as the use of smart contracts increases.

1.4 Thesis Statement

Cloud-based automated machine learning can assist organizations with rapidly identifying vulnerable smart contracts.

Using cloud-based AutoML for smart contract vulnerability detection brings the power of machine learning to address this contemporary challenge without requiring specialized machine learning skills, expertise, or the steep upfront expense and effort associated with setting up specialized computer hardware and software. The use of this technology should lead to an increase in the timely detection of vulnerable smart contracts while lowering the chances of deploying smart contracts with detectable flaws on the operational blockchain platform.

1.5 Research Objectives

The primary objectives of this research are as follows:

1. Use cloud-based automated machine learning to build vulnerability detection models that identify smart contracts with flaws.
2. Use smart contract source code in its native form as a basis to build and evaluate vulnerability detection models.
3. Determine whether models developed using cloud-based automated machine learning can identify vulnerable smart contracts with an acceptable level of performance.
4. Study the effectiveness of cloud-based automated machine learning models in identifying common types of smart contract vulnerabilities.

1.6 Research Questions and Hypotheses

The research questions and hypotheses for this study are expressed in the following subsections.

1.6.1 Research Questions

RQ1: Can cloud-based automated machine learning be used to build effective models to detect smart contract vulnerabilities?

RQ2: Can smart contract source code in its native form without an intermediate representation be used to build effective vulnerability detection models?

RQ3: How quickly can smart contract vulnerability detection models built using cloud-based automated machine learning identify vulnerable smart contracts?

RQ4: Are there advantages to using specific types of compute capabilities when using cloud-based automated machine learning to build and test smart contract vulnerability detection models?

1.6.2 Hypotheses

H1: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with at least one of the common types of vulnerabilities with accuracy greater than 80% and precision greater than 80%.

H2: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as

insecure arithmetic flaws with accuracy greater than 80% and precision greater than 80%.

H3: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as block values as a proxy for time flaws with accuracy greater than 80% and precision greater than 80%.

H4: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as reentrancy flaws with accuracy greater than 80% and precision greater than 80%.

H5: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as locked Ether flaws with accuracy greater than 80% and precision greater than 80%.

H6: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as transaction order dependency flaws with accuracy greater than 80% and precision greater than 80%.

1.7 Scope of Research

This research studies the use of cloud-based AutoML to build binary classification models that identify vulnerable smart contracts based on smart contract source code in its native form. Data from the Ethereum blockchain and the ScrawlD dataset are used to build and test the AutoML models. The ScrawlD dataset consists of

9,252 Ethereum smart contract addresses, along with a mapping of any vulnerabilities identified using traditional smart contract testing tools (Yashavant et al., 2022). The software source code for these 9,252 smart contracts was obtained from the Ethereum blockchain and labeled using the ScrawlD vulnerability dataset. This research examines five smart contract vulnerability types: insecure arithmetic, block values as a proxy for time, reentrancy, locked Ether, and transaction order dependency flaws. The research also explores the use of graphics processing unit-based and central processing unit-based compute capabilities within a cloud-based AutoML platform to build and evaluate models. The AutoML-built models are evaluated using statistical performance metrics to determine their effectiveness. Of particular interest are the accuracy and precision metrics.

1.8 Research Limitations

This research focuses on the use of cloud-based AutoML capabilities and does not explore other machine learning approaches and platforms. In particular, Microsoft's Azure AutoML is used for the testing part of this praxis. This study is limited to smart contracts on the Ethereum network, programmed using the Solidity language. Specifically, the smart contracts compiled in the ScrawlD study are used. Other datasets and smart contract programming languages are not examined. The research also focuses on the use of native smart contract source code to build machine learning-based vulnerability detection models. It does not examine intermediate source code representations such as abstract syntax trees, opcode, and bytecode. The research is also limited to five common smart contract vulnerability types and does not evaluate the numerous other known types of vulnerabilities.

1.9 Organization of Praxis

This praxis is organized into five chapters. Chapter 1 introduces the research topic, research motivations, problem statement, research questions, hypothesis, research scope, and limitations. A literature review covering blockchain, smart contracts, smart contract vulnerabilities, vulnerability detection, machine learning, and AutoML is provided in Chapter 2. Chapter 3 describes the methodology used to carry out this research. The research results are presented in Chapter 4. Chapter 5 covers the verification of hypotheses, responses to research questions, contributions to the body of knowledge, and future research recommendations.

Chapter 2—Literature Review

2.1 Introduction

This chapter explores the literature on blockchain smart contracts, smart contract vulnerabilities, cloud-based automated machine learning (AutoML), and smart contract vulnerability detection. Section 2.2 provides a brief background on blockchain smart contracts, their origins, the popular Ethereum network, and the general characteristics of smart contracts. Section 2.3 reviews smart contract vulnerabilities, root causes, and challenges and provides an overview of the five common smart contract vulnerabilities examined in this research. Sections 2.4 and 2.5 provide an overview of machine learning and cloud-based AutoML. Section 2.6 reviews current approaches to smart contract vulnerability detection and discusses the challenges and limitations of current methods. Finally, Section 2.7 summarizes the literature review, the gaps identified, and the new insights that this praxis introduces to the body of knowledge.

2.2 Blockchain Smart Contracts

Nick Szabo minted the term *smart contract* in the 1990s for "a set of promises, specific in digital form, including protocols within which the parties perform on those promises" (Szabo, 1996, p. 1). Andrew Miller defines smart contracts as "programs that run atop of a financial infrastructure, and command the flow of money according to user-defined rules" (Miller, 2018). Blockchain can be regarded as the modern financial infrastructure that enables smart contracts (Zheng et al., 2020).

Blockchain is a distributed ledger technology that allows digital transactions to be recorded and executed in a decentralized manner using cryptographically immutable blocks and a consensus protocol (Zheng et al., 2018). Blockchain became renowned for

its role as the enabling technology of Bitcoin (Nakamoto, 2008). Since then, however, numerous other public and private solutions have been built on blockchain technology and concepts. One such popular solution is the Ethereum, a decentralized blockchain platform. Ethereum was launched in 2015 with a focus on enabling blockchain smart contracts (Chen et al., 2020). Ethereum's founder, Vitalik Buterin, envisioned smart contracts as supporting financial applications in subcurrencies, financial derivatives, savings wallets, wills, and employment contracts (Buterin, 2014).

There are many benefits to using smart contracts over traditional ones. According to Zheng et al. (2020), the immutable property of smart contracts greatly helps reduce fraud risk with financial transactions. They also state that smart contracts can greatly reduce administrative and service costs by eliminating the need for a third-party intermediary. Furthermore, eliminating third-party intermediaries helps significantly improve business process efficiency (Zheng et al., 2020).

Smart contracts are not without limitations, however. The immutable nature of smart contracts presents several operational challenges. According to Nzuva (2019), traditional contracts are often amended to address the latest needs and changing environmental conditions. Such changes are impossible to make with smart contracts, however, given that smart contracts are by design not updatable once placed into operational use (Nzuva, 2019). Smart contracts also lack privacy due to the public nature of platforms such as Ethereum. Moreover, Nzuva (2019) states that legal enforceability is another challenge with smart contracts. Most laws and regulations are written to address traditional contracts, and there is limited precedent regarding their applicability and

support for smart contract-based transactions. Given these challenges, it is essential to carefully evaluate smart contracts for suitability before using them.

Smart contracts on Ethereum can be developed using a variety of programming languages. The most widely used language is Solidity, a Turing-complete object-oriented programming language (Wöhrer & Zdun, 2018). Solidity is popular because of its similarity to the syntax of JavaScript, a well-known webpage scripting language. The typical Ethereum smart contract is between 100 and 200 lines of code (Pinna et al., 2019). A simple example of the syntax of Solidity is shown in Figure 2. In this example, the value held in the storedMoney variable is returned when the function is called.

```
1  pragma solidity ^0.8.4;
2
3  contract HelloMoney {
4    uint storedMoney;
5    function set(uint x) public {
6      storedMoney = x;
7    }
8    function get() public view returns (uint) {
9      return storedMoney;
10   }
11 }
```

Figure 2—Example Solidity Source Code

Solidity source code is compiled into bytecode that is then executed on a decentralized virtual computer called the Ethereum Virtual Machine (EVM; Pinna et al., 2019). When the contracts are executed on Ethereum, they are assigned a 160-bit identifier, typically referred to as the smart contract address (Luu et al., 2016). Once a smart contract is deployed on Ethereum, the underlying source code cannot be modified (Ferretti & D'Angelo, 2019). This is why it is crucial to detect vulnerabilities during the development stage of smart contracts.

2.3 Smart Contract Vulnerabilities

Since their introduction, Ethereum smart contracts have been riddled with vulnerabilities. Luu et al. (2016) completed one of the earliest studies on smart contract vulnerabilities, revealing that approximately 45% (8,833 of the 19,366) of the tested smart contracts had security vulnerabilities. This is an alarming percentage, given that billions of US dollars are stored and transferred using Ethereum smart contracts (Miller et al., 2018).

In their survey of smart contract attacks, Atzei et al. (2017) highlight several reasons why Ethereum smart contracts are prone to flaws. They state that one of the primary causes is the smart contract development language. Solidity is not a user-friendly language and often leads developers to make incorrect assumptions due to its semantic similarity with other more mature languages such as JavaScript (Atzei et al., 2017). Another reason, according to Tikhomirov et al. (2018), is the fast-paced nature of smart contract development and deployment cycles, which limits the time available for testing and other quality-control activities.

With traditional software technologies, once defects are uncovered, the developers can generally develop and release patches to address the discovered issues. Given the immutable nature of smart contracts, however, source code cannot be modified once deployed on the Ethereum network, making it impossible to fix discovered vulnerabilities (Dika & Nowostawski, 2018). As a result, it is critical to uncover any flaws before deploying smart contracts.

Five common Ethereum smart contract vulnerabilities are insecure arithmetic, block values as a proxy for time, reentrancy, locked Ether, and transaction order dependency.

2.3.1 Insecure Arithmetic

According to the Smart Contract Weakness Classification Registry (2023), insecure arithmetic, sometimes referred to as integer overflow and underflow, occurs when numerical operations in smart contracts result in values that fall outside the allowed range of an integer variable. When the value exceeds the maximum of the range, the value tends to revert to zero. This is referred to as an integer overflow. When the value falls below the minimum of the range, the value may be elevated back to the maximum of the range. This is referred to as an integer underflow.

Insecure arithmetic flaws in smart contracts can be invoked accidentally or intentionally manipulated by a malicious hacker during the execution of a smart contract to steer the outcome of transactions. Depending on the variables they affect, insecure arithmetic flaws can be exploited to transfer smart contract assets to unintended parties or cause contracts to prematurely execute because of affected timing functions. This type of vulnerability has various other significant consequences.

Pandey (2023) states that insecure arithmetic flaws can be prevented by using proper datatypes and incorporating automatic checks for overflow and underflow before executing arithmetic operations. SafeMath is a readily available Solidity library that includes a set of functions that perform arithmetic operations and automatically checks for overflow and underflow issues (Pandey, 2023). Developers should leverage

capabilities such as SafeMath to ease smart contract development and minimize the risk of insecure arithmetic flaws.

2.3.2 Block Values as a Proxy for Time

The Smart Contract Weakness Classification Registry describes block values as a proxy for time flaws as those resulting from the use of block timestamps and block numbers as a part of timing functions within smart contracts (Smart Contract Weakness Classification Registry, 2023). Since the block timestamp is usually based on the local node clock, which is configurable by the node user, it can be manipulated by a malicious user (Wu et al., 2023). A malicious user may be able to steer the execution of a transaction by simply varying their local node clock, potentially providing them with an unfair advantage during the transaction. Similarly, block numbers can also be used to predict time differences (Smart Contract Weakness Classification Registry, 2023). Luu et al. (2016) recommend that developers use a non-alterable source instead of block timestamps to avoid the vulnerability.

2.3.3 Reentrancy

Smart contracts often directly interact with other smart contracts to carry out transactions. Usman et al. (2021) state that a reentrancy flaw occurs when one of the smart contracts is able to invoke or call functions in another smart contract prior to the smart contract updating its state information (i.e., account balances) from the prior transaction. The reentrancy flaw could enable a malicious user to withdraw more value than available in their account balances. When these calls are performed recursively, this can have a devastating effect on the exploited smart contract. For example, during the infamous 2016 decentralized autonomous organization attack, hackers exploited a

reentrancy flaw to drain the organization's smart contracts of $60 million (Alchemy, 2022).

According to the Smart Contract Weakness Classification Registry (2023), however, the risk of reentrancy flaws can be minimized by ensuring that all the internal state information of smart contracts is updated before executing external function calls.

2.3.4 Locked Ether

Locked Ether occurs when the assets held by a smart contract are not accessible and become unusable. This flaw can occur when the smart contract is built to accept Ether, but the developer fails to incorporate a proper withdrawal or transfer function. According to Perez and Livshits (2019), locked Ether can also occur when the withdrawal or transfer function depends on an external smart contract and that external contract has been destroyed or is otherwise no longer functional. In essence, the effects of locked Ether are analogous to a denial of service attack on the smart contract.

Locked Ether flaws can be prevented by ensuring that a functional Ether withdrawal or transfer mechanism is included when developing smart contracts with functions with accept Ether. In addition, Usman et al. (2021) state that one approach to minimizing the risk of locked Ether flaws is to perform address checks of methods that support external call functions related to the withdrawal or transfer of assets.

2.3.5 Transaction Order Dependency

Transaction order dependency flaws are essentially race conditions within smart contracts that occur when transactions are added to the blockchain. A race condition occurs in computing when the intended order of computational operations is bypassed or otherwise manipulated, leading to unintended outputs or access. According to Heimbach

and Wattenhofer (2022), when a transaction on the Ethereum network is ready to be added to the block, it is broadcast across the network and placed in a transaction queue called mempool. Since Ethereum is a public network, other nodes can observe the contents of the transaction in mempool prior to it being added to the block. This exposure allows a malicious node to read the transaction output and submit their own transaction with the stolen output for addition to the network. Chibuzor (2022) states that for the malicious node to prioritize its transaction over the original transaction, the malicious node would simply have to advertise a higher gas price (miner reward) than the original node. This results in the reordering of the new malicious transaction over the original transaction, thereby allowing the malicious node to claim the reward.

According to the Smart Contract Weakness Classification Registry (2023), transaction order dependency flaws can be avoided by implementing a commit reveal hash scheme within the smart contract. A commit reveal hash scheme involves the original node submitting a hash of a salt, node address, and answer, to the smart contract, instead of exposing the actual answer (Smart Contract Weakness Classification Registry, 2023). The smart contract would then store the hash and the sender's address. Once ready to claim the reward, the original node would then submit the salt and the answer (Smart Contract Weakness Classification Registry, 2023). The smart contract would then compute the hash using the salt, answer, and sender's address. If the computed hash matches the stored hash, the reward is released.

2.4 Machine Learning

In the introduction to his 1997 book on machine learning, Tom Mitchell says that "ever since computers were invented, we have wondered whether they might be made to

learn. If we could understand how to program them to learn—to improve automatically with experience—the impact would be dramatic" (Mitchell, 1997). Thanks to advances in machine learning technology, this is now becoming a reality. Machine learning is a subfield of artificial intelligence focused on the use of data and algorithms by computers to "learn" and build models (IBM, 2023). Machine learning models can process all types of data, including numbers, text, images, and videos. One of the primary approaches is supervised machine learning.

Supervised machine learning is an approach in which the computer is provided with labeled datasets for the purposes of training a prediction model. The labeled datasets include the correct prediction (dependent variable) along with the independent variables associated with that prediction. The computer analyzes all the independent variable values along with their respective dependent variable values and develops a mapping function to provide a prediction based on a new set of independent variable values (Kavitha, 2021). The resulting algorithm, optimized based on the training, is referred to as a machine learning model. Two primary types of supervised learning are classification and regression. Classification is associated with predicting discrete or categorical values, and regression is associated with the prediction of continuous values (Kavitha, 2021).

Text classification is a type of supervised machine learning technique that deals with data such as source code. Assessing whether a particular source code is vulnerable—a binary determination—would be considered a binary classification task. Computers must convert text data into a numerical form to effectively build a model for text classification (Kavitha, 2021). According to Kavitha et al. (2021), steps such as tokenization, vectorization, a variety of different encodings and word counts, and topic

analysis are used to transform text data into a format that is computable. While many of the processing steps can be handled by contemporary algorithms, there is still a need for substantial expertise and effort to properly preprocess the data, develop machine learning scripts, and tune a model.

2.5 Cloud-Based Automated Machine Learning

Chen et al. (2019) postulate that AutoML strives to replace the role of humans in some or all of the configuration related work with traditional machine learning. (Chen et al., 2019). The AutoML field has substantially evolved during the past decade, and there are now several toolkits and platforms that users with limited machine learning knowledge can use (Escalante, 2020). Hanussek et al. (2020) note in their evaluation of AutoML that it is generally capable of outperforming or achieving results similar to those produced by human data scientists using nonautomated machine learning approaches. There are several AutoML platforms, some of which are shown in Figure 3.

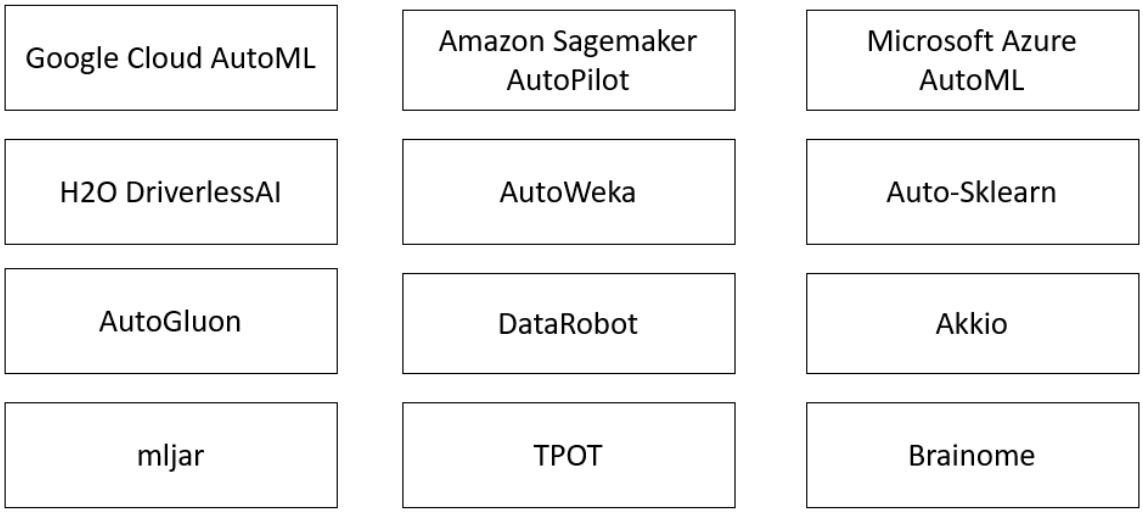

Figure 3—Popular AutoML Platforms

Most AutoML platforms follow a similar sequence of steps to build a model in an automated manner with minimal code or human involvement. Figure 4 shows the primary AutoML steps: data preprocessing, feature engineering, model selection, hyperparameter

optimization, and model evaluation (Truong et al., 2019). This level of automation allows teams to rapidly employ machine learning solutions while allowing humans to focus on problem definition, data collection, and model deployment (Chen et al., 2019).

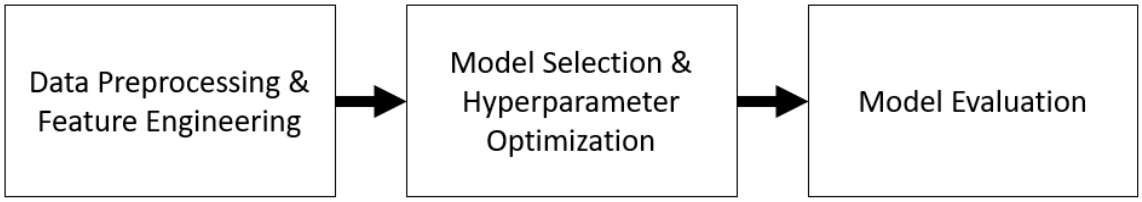

Figure 4—Overview of Primary AutoML Steps

Several large cloud providers now offer AutoML as a service (Truong et al., 2019). Cloud-based AutoML services further lower the entry bar to access machine learning capabilities, as they eliminate the need for the user to purchase and install expensive computing hardware and to install and maintain sophisticated software packages necessary for machine learning. A study in 2021 showed that cloud-based machine learning platforms significantly reduced the need for costly local machine learning capabilities (Noshiri et al., 2021). The cloud-based approach allows for ubiquitous, convenient, and on-demand access that can be allocated and released with limited effort and allows the users to only pay for what they actually use (Mel & Grance, 2011). Users typically interact with a cloud-based machine learning platform through a web interface (Noshiri et al., 2021).

2.5.1 Data Preprocessing and Feature Engineering

Real-world data sources are often incomplete, inconsistent, and riddled with errors. During data preprocessing, AutoML cleans, formats, and organizes raw real-world data so that the machine learning models can be better trained (Cysneiros Aragao et al., 2023). Any missing values are handled by injecting a substitute value or by removing the associated record from the dataset (Microsoft, 2023). During this stage, techniques can be

employed to address any data imbalance issues in addition to ensuring that the data is appropriately scaled or normalized (Microsoft, 2023).

Once data preprocessing is complete, AutoML performs feature engineering to select key original features and build new features based on transformations applied to the original features to help differentiate the patterns in the dataset (Waring et al., 2020). For example, Azure AutoML uses a text featurizer to convert vectors of text tokens into sentence vectors during the feature engineering step when the dataset includes textual data (Microsoft, 2023).

2.5.2 Model Selection and Hyperparameter Optimization

Model selection is the step in which the appropriate machine learning algorithm for training is identified based on the dataset. Traditionally, this was performed by humans based on experience and expertise, often using a trial-and-error approach (Yao et al., 2019). AutoML automates this process by selecting a set of potential algorithms from an existing pool of known algorithms based on the dataset, features, and type of machine learning task (classification, regression, etc.) and then building corresponding models. As models are built and trained, their ability to fit the data is monitored by AutoML, and the system iteratively down-selects until the best model(s) is identified based on the desired performance metric objectives.

Model selection also involves hyperparameter optimization. Machine learning models rely on values known as hyperparameters to control the behavior of the algorithm and training process (Waring et al., 2020). Hyperparameter optimization is the process by which the best values are selected. With standard machine learning, this is a time-consuming process that requires expertise and experience to complete (Waring et al.,

2020). AutoML automates the process by which these parameters are set and optimized using a variety of algorithms (Cysneiros Aragao et al., 2023).

2.5.3 Model Evaluation

The final step in AutoML is model evaluation. In this step, the model is tested using a test dataset. Testing allows for the generation of performance metrics such as accuracy, precision, recall, F1 score, Matthews correlation coefficient, and duration. These metrics provide an understanding of the effectiveness of the model.

2.6 Smart Contract Vulnerability Detection

Effectively detecting vulnerabilities in smart contracts before deployment to the production blockchain network is essential to minimizing security risks to the assets they store and transfer. Several approaches to the security evaluation of smart contracts are available to assist with detecting vulnerabilities. Dika and Nowostawski (2018) highlight formal security audits by seasoned blockchain developers and specialized teams as one of the most effective ways to discover vulnerabilities in smart contracts. This approach is time-consuming and costly, however (Dika and Nowostawski, 2018). Other approaches include the use of static and dynamic code analysis tools. Static code analysis tools examine smart contract source code, bytecode, or opcode for issues. Static code analysis tools detect vulnerabilities by examining the code without executing it (Feist et al., 2019). Dynamic code analysis tools execute the smart contract in a runtime environment and evaluates its behavior based on input data (Tihkomirov et al., 2018).

2.6.1 Detection Using Static Code Analysis Tools

Feist et al. (2019) consider static analysis to be one of the most effective ways of detecting vulnerabilities in smart contracts. Feist et al. (2019) describe the ideal

properties for an effective static code analysis framework. These include the correct level of abstraction, robustness, performance, and accuracy. The correct level of abstraction ensures that the analysis tool is at the right level to ensure optimal vulnerability detection. Robustness ensures that the analysis framework can handle real-world code without crashing. Performance refers to the speed of the analysis. Finally, accuracy considers how effective the framework is in detecting issues while ensuring that the false-positive rate is low (Feist et al., 2019).

Tikhomirov et al. (2018) introduced a static code analysis tool called SmartCheck that parses smart contract source code to develop an intermediate representation and then search for vulnerabilities based on a database of known vulnerability patterns (Tikhomirov et al., 2018). Their testing of 4,600 Ethereum smart contracts using SmartCheck revealed that 63.8% of the contracts had critical vulnerabilities (Tikhomirov et al., 2018). Another static code analysis tool, Slither, was introduced by Feist et al. in 2019. Like SmartCheck, Slither relies on developing an intermediate representation of the smart contract code (Feist et al., 2019). While both SmartCheck and Slither can detect several common flaws, including the vulnerability associated with the famous 2016 decentralized autonomous organization attack, Slither was shown to have higher performance and accuracy than SmartCheck (Feist et al., 2019).

Several studies have also established acceptable thresholds for statistical measures associated with effective vulnerability detection. Qian et al. (2022) established a high accuracy score as one above 80%. Their work also explored detection time periods. They noted that static code analysis tools took on average anywhere from 2.1 seconds to 352.2 seconds to detect vulnerabilities (Qian et al., 2022). Static analysis methods rely on the

definition of matching rules based on patterns of known vulnerabilities to identify flaws with smart contracts (Xu et al., 2021). As a result, traditional static analysis can be costly and time consuming (Gao et al., 2019).

2.6.2 Detection Using Dynamic Code Analysis Tools

Dynamic code analysis tools use symbolic execution, taint tracking, and fuzzing techniques during the operation of smart contracts to detect flaws (Feist et al., 2019).

One of the earliest studies on smart contract vulnerability detection was conducted by Luu et al. in 2016. They completed a review of known security vulnerabilities and introduced an analysis tool called Oyente to help developers detect vulnerabilities. Oyente is considered one of the earliest smart contract dynamic analysis security tools. It evaluates the EVM bytecode for known defects within smart contracts (Luu et al., 2016). Their testing of 19,366 smart contracts using Oyente found that 45% of the smart contracts had at least one security vulnerability (Luu et al., 2016). Another leading dynamic security analysis tool is Mythril (Mueller, 2018). Similar to Oyente, Mythril evaluates the EVM bytecode for known vulnerabilities by analyzing all probable program execution states (Sharma, 2022).

2.6.3 Detection Using Machine Learning

In 2019, Momeni et al. introduced a machine learning model for detecting vulnerabilities in smart contracts. To train their models, they first obtained smart contracts from the Ethereum network and tested them using the static code analysis tools Slither and Mythril (Momeni et al., 2019). They also extracted an abstract syntax tree as an intermediate representation for each of the smart contracts (Momeni et al., 2019). Using the results from the static code analysis tools, they then labeled each of the smart

contract abstract syntax trees to indicate whether the code contained each of the 46 vulnerabilities that they had tested for. These data were then split into training and test datasets, with the training dataset being used to train several machine learning models. The machine learning models were then tested using the test dataset and evaluated based on accuracy, recall, and precision. Momeni et al.'s work revealed that a machine learning model could predict 16 different vulnerabilities with an average accuracy rate of 95%. Furthermore, they demonstrated that a machine learning model was exponentially faster at detecting smart contract vulnerabilities than static code analysis tools such as Mythril and Slither (Momeni et al., 2019).

Wang et al. (2019) proposed a machine learning-based smart contract vulnerability detection tool called ContractWard. Like Momeni et al. (2019), their work relied on extracting smart contracts from the Ethereum network, testing and labeling the datasets using static code analysis tools, and then training machine learning models using the dataset. However, Wang et al. (2019) used operation codes (opcodes) instead of abstract syntax trees extracted from source code. Moreover, they used a variety of oversampling techniques to correct imbalances within the datasets. Their testing also demonstrated that machine learning models could detect vulnerabilities with a high degree of precision and recall while being substantially faster than traditional tools (Wang et al., 2019).

Eth2Vec, a tool proposed by Ashizawa et al. in 2021, uses a neural network for natural-language processing to automatically extract features from EVM bytecodes. Eth2Vec was confirmed by the researchers to outperform the earlier machine learning models, such as the one proposed by Momeni et al. in 2019 (Ashizawa et al., 2021).

Another machine learning-based tool using a natural-language processing model was proposed by Jeon et al. in 2021. This tool, named SmartConDetect, is based on a Bidirectional Encoder Representations from Transformers model trained using a dataset with smart contract code fragments associated with vulnerabilities (Jeon et al., 2021). Testing using this tool also proved that a natural-language processing-based model could detect smart contract vulnerabilities with a high degree of precision and recall.

Wartschinski et al. (2022) explored the use of raw source code to build vulnerability detection models capable of detecting flaws in Python code using machine learning. Their work recognized precision scores of 80–90% as acceptable (Wartschinski et al., 2022).

In 2022, Zeng et al. proposed EtherGIS, a smart contract vulnerability detection tool that uses graph neural networks to extract graph features from EVM bytecode. They also used AutoML to assist with model training (Zeng et al., 2022). Specifically, AutoML was used for key aspects such as feature engineering and hyperparameter optimization (Zeng et al., 2022).

2.7 Summary and Conclusion

This literature review has shown that vulnerabilities in blockchain smart contracts are a serious and increasing problem. While several tools and approaches have been introduced in the past few years to help address the need for effective vulnerability detection, it remains a challenge. In their study on the security perspectives of smart contract developers, Wan et al. (2021) highlighted that only 54% of the developers indicated that they used security tools in their development process. In other words, 46% of developers are not using what is generally available today. This could be because the

toolsets that are available are cumbersome, complex, time-consuming, or just not readily available for use by the general development community. The current toolset challenges highlight the need for a more readily accessible smart contract vulnerability detection solution.

While there has been recent work on using machine learning techniques to detect vulnerabilities more quickly and accurately with blockchain smart contracts, traditional machine learning is not accessible to many people due to the need for specialized data science and machine learning skills and the upfront costs and effort necessary to set up the necessary hardware and software. Cloud-based AutoML can ease many of these constraints.

In addition, the machine learning-based vulnerability detection approaches discussed in this literature review focused on models built using intermediate representations or another processed view of the source code. The literature review did not identify any evaluation of a model trained with native smart contract source code. As smart contract source code tends to be condensed in nature—typically between 100 and 200 lines of code (Pinna et al., 2019)—there may be an opportunity to directly use the native code without an intermediate representation to build and train detection models.

Given the gaps identified in this literature review, this praxis focuses on studying the effectiveness of using cloud-based AutoML to build models using smart contract source code in its native form that can reliably detect whether a smart contract has common vulnerabilities. Specifically, the research focuses on building and evaluating binary classification models to identify whether a smart contract is susceptible to insecure

arithmetic, block values as a proxy for time, reentrancy, locked Ether, and transaction order dependency flaws.

Chapter 3—Methodology

3.1 Introduction

This chapter discusses the methodology used to carry out the research performed in this praxis. The primary goal of this research is to study the use of models built using cloud-based automated machine learning (AutoML) and smart contract source code in its native form, to detect common vulnerabilities in smart contracts, answer the research questions, and verify the hypotheses expressed in Chapter 1. Figure 5 gives a high-level overview of the methodology.

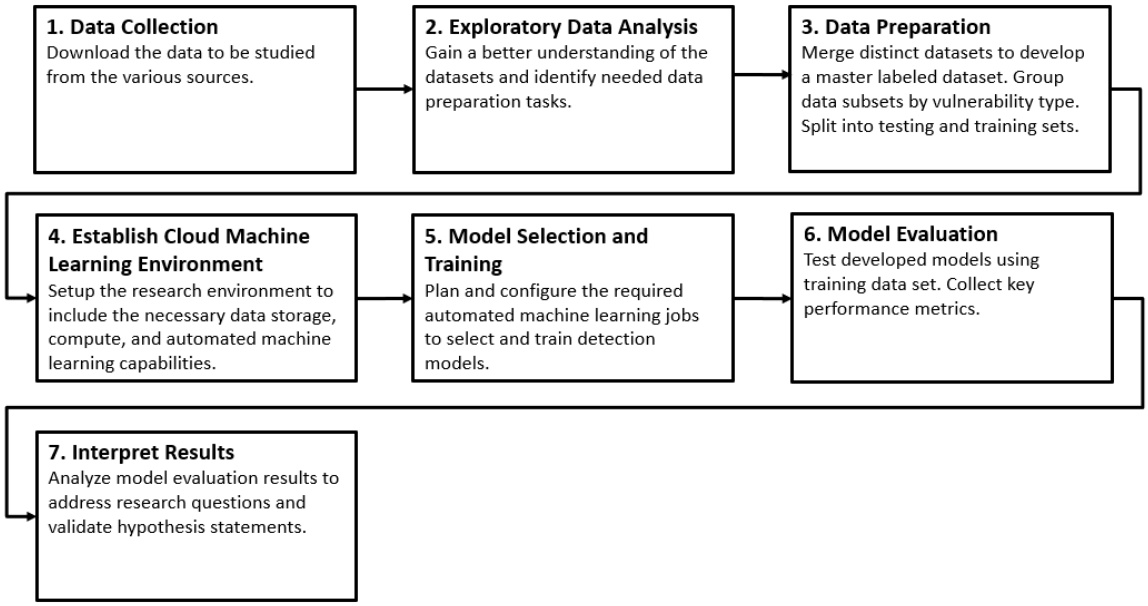

Figure 5—Research Methodology Overview

Section 3.2 describes the data sources, collection steps, data analysis, and preparation method. In Section 3.3, the AutoML platform and research environment setup are discussed. Section 3.4 covers experiment design and the selection and training of models using AutoML. Finally, Section 3.5 discusses the performance metrics used to evaluate the models.

3.2 Data Collection and Analysis

Machine learning relies on datasets to train models. To train and test AutoML models that can detect the corresponding smart contract vulnerabilities, this research requires smart contract source code data labeled with any known vulnerabilities. The literature review revealed that smart contract vulnerability datasets are available for researchers to use for training and testing models.

3.2.1 Data Sources

One of the more recently released datasets is ScrawlD. The ScrawlD dataset was published by Yashavant et al. in 2022 to assist researchers in vulnerability classification and detection. The dataset is based on vulnerability assessments conducted on real-world Ethereum smart contracts using Slither, SmartCheck, Mythril, Oyente, and Osiris (Yashavant et al., 2022). The work focused on several types of vulnerabilities, including insecure arithmetic, locked Ether, reentrancy, block values as a proxy for time, and transaction order dependency (Yashavant et al., 2022). The original ScrawlD dataset covered 6,780 smart contracts. This dataset identified the smart contract address and listed the vulnerabilities detected within the smart contract. Shortly after their paper introducing the ScrawlD dataset was published, Yashavant et al. (2022) released an updated dataset. This updated dataset covers 9,252 unique smart contracts and is used in this praxis.

Since the ScrawlD dataset only includes the smart contract address and detected vulnerabilities, the smart contract source code necessary to train and test machine learning models must be obtained separately. The source code for Ethereum contracts is

generally made available to the public and is accessible via tools such as Etherscan (Pinna et al., 2019).

3.2.2 Data Collection

The ScrawlD dataset is available for download via GitHub. Two data files were obtained from the ScawlD dataset for this praxis. The first was the listing of the Ethereum smart contract addresses for the 9,252 smart contracts tested by Yashavant et al. (2022). This file is available in a comma-separated-values format. The second was a text file that contains a listing of the subset of smart contract addresses that were deemed to be vulnerable, along with the detected vulnerabilities.

The smart contract source code used in this praxis was downloaded using Etherscan. Etherscan is a block explorer that allows users to search and browse Ethereum blockchain data and the underlying source code of many smart contracts (Etherscan, 2023). In addition to providing access to Ethereum information through an interactive web page, Etherscan provides application programming interface (API) capabilities to facilitate automated access. Access to the Etherscan API requires the creation of an account and API keys with Etherscan. An API account was established to support the data collection for this praxis. This API account, along with the latest ScrawlD smart contract address list and a Python script made available by Yashavant et al. (2022) as part of their ScrawlD effort, was used to download data including the source code for 9,252 smart contracts via Etherscan. Information associated with each smart contract was downloaded as a distinct JavaScript Object Notation formatted file. For ease of use, once downloaded, the 9,252 individual JavaScript Object Notation files were compiled into a single comma-separated-values file.

3.2.3 Exploratory Data Analysis

Exploratory data analysis (EDA) refers to the process that researchers often follow early in their investigation to better understand the dataset, identify patterns, and detect inconsistencies, gaps, and other issues with the help of statistics and visualizations (Patil, 2018). In this praxis, Exploratory Desktop was used to conduct an EDA on the collected data. Exploratory Desktop is built upon the open-source R statistical software and includes functionality that supports data visualization, data wrangling, statistical analysis, and machine learning via a user-friendly graphical user interface (Exploratory, 2023).

First, Exploratory Desktop was used to review the smart contract dataset that was downloaded using the Etherscan API. This allowed for a better understanding of the data fields (often referred to as features in machine learning vernacular) in the dataset. The EDA provided insights into the relevance of the different data fields to this research.

Next, EDA was conducted on the listing of vulnerable smart contracts. The EDA revealed the distribution of the various types of vulnerabilities in the dataset. This information was used to further identify the vulnerability data needed for this research. In addition, any class imbalances can be observed during the EDA process. It is prudent to address any observed class imbalances during the data preparation stage since class imbalances can greatly affect the performance of machine learning models.

3.2.4 Data Preparation

With the insights gained from the EDA, the data were organized and prepared for experimentation. The first step in the data preparation was to label the downloaded smart contract source code data using the ScrawlD vulnerability dataset. The smart contract

address is a unique identifier present in both datasets. A table join is performed to merge the two datasets using the smart contract address as the key. Next, given the focus of this praxis on the use of smart contract source code to build detection models, all other data fields were removed from the downloaded data. Only the fields containing the smart contract source code, smart contract address, and detected vulnerabilities are retained. Finally, the selected data fields were reviewed for completeness to ensure that any rows with missing data fields are removed. The resulting output served as the master dataset for this research.

The master dataset was then used to create six new data subsets based on the vulnerability types planned for study. Five of the datasets each focus on a specific vulnerability type. The vulnerability types are insecure arithmetic, block values as a proxy for time, reentrancy, locked Ether and transaction order dependency flaws. With each of these datasets, any rows containing smart contract source code that contains the respective dataset vulnerability are labeled under a field titled “Vulnerable” as “YES.” Those without the vulnerability are labeled as “NO.” The sixth dataset is a binary labeled set that was created to train a model to provide a basic binary determination of whether a particular smart contract contains any of the known vulnerabilities. This sixth dataset is referred to hereafter as the VulnerableOrNot dataset.

Once all six datasets were established, they were split into training and testing subsets, with 80% of the data allocated for the training and 20% set aside for the testing. After this split, there were 12 discrete datasets. Figure 6 shows the dataset preparation steps and resulting six training and six testing datasets.

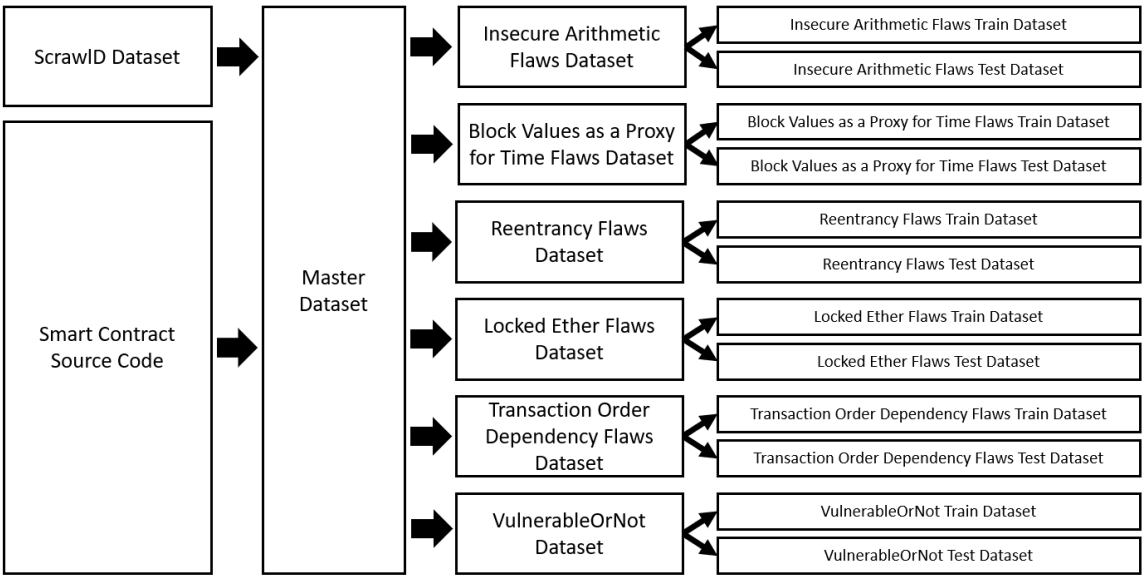

Figure 6—Training and Testing Dataset Preparation Sets

3.3 Machine Learning Environment

According to Gartner's Magic Quadrant report for Cloud Infrastructure and Platform Services, the three leading cloud service providers are Amazon Web Services, Microsoft Azure, and Google Cloud. All three offer AutoML services. Microsoft Azure was selected as the cloud-based AutoML platform for this research, as the process for obtaining access to the platform for academic research purposes was the most streamlined. Microsoft Azure's AutoML service is marketed as Azure AutoML. Azure AutoML enables users to create complex machine learning models with basic data science and machine learning expertise (Microsoft, 2023).

Azure AutoML evaluates several algorithms for model selection. For classification tasks, the following algorithms are considered:

- Logistic Regression
- K-Nearest Neighbors
- Extreme Gradient Boosting (XGBoost)
- Stochastic Gradient Descent

- Decision Tree Linear
- Support Vector Machine
- Multinomial Naïve Bayes
- Random Forest
- Averaged Perceptron Classifier
- Bernoulli Naïve Bayes
- Extremely Randomized Trees
- Support Vector Machine
- Light Gradient-Boosting Machine (LightGBM)

In addition, when dealing with text data, Azure AutoML transforms the data into numbers and vectors as part of the featurization step, which often involves the use of scalers and normalizers. This step helps the models better learn the patterns within the data as part of model training.

Access to Azure AutoML was obtained by first creating a Microsoft Azure subscription account. Then, a new resource group was created. A resource group is a virtual grouping within Azure that holds all the resources used for a particular effort. (Microsoft, 2023). It allows for ease of organization and management of related resources. Next, within the resource group, an Azure Machine Learning workspace was created. An Azure Machine Learning workspace is a centralized space within a resource group that helps manage and track all machine learning space-related items, including network settings, security permissions, and activity logs. Within the Azure Machine Learning workspace, the Azure Machine Learning Studio is used for machine learning

tasks, including creating compute clusters, uploading datasets, configuring AutoML jobs, evaluating models, and generating performance metrics.

Azure AutoML requires the provisioning of compute resources to perform its work. For this research, dedicated compute clusters were used. Compute clusters are single or multi-node virtual servers that can be provisioned within the Azure Machine Learning Studio. These clusters are available with central processing unit (CPU) and graphics processing unit (GPU) processors and a variety of memory and storage options. Both CPUs and GPUs were used in this praxis to explore whether there are differences in efficiency when using one or the other.

The GPU cluster used in this praxis was equipped with an NVIDIA Tesla K80 GPU, 56 GB RAM, and 380 GB disk space. The CPU cluster was configured with an Intel CPU, 14 GB RAM, and 56 GB disk space. At the time of the research, CPU and GPU cluster usage were priced at $0.23 per hour and $0.90 per hour, respectively.

3.4 Model Selection and Training

Once the environment was set up, it was ready to handle AutoML workloads. First, the datasets were uploaded into the testing environment. Next, experiments were set up to test each of the hypotheses. With each experiment, two different AutoML jobs were set up. One job was configured to use the GPU cluster, and the other was set up to use the CPU cluster. In total, 12 jobs were set up covering six experiments. Table 2 provides an overview of the 12 jobs and associated experiments and compute cluster type.

Experiment	Compute Cluster Type	Job
VulnerableOrNot	CPU	1
	GPU	2
Insecure Arithmetic Flaws	CPU	3
	GPU	4
Block Values as a Proxy for Time Flaws	CPU	5
	GPU	6
Reentrancy Flaws	CPU	7
	GPU	8
Locked Ether Flaws	CPU	9
	GPU	10
Transaction Order Dependency Flaws	CPU	11
	GPU	12

Table 2—Organization of Experiments

All the experiments were configured as binary classification jobs. AutoML requires the selection of a primary optimization metric. Microsoft recommends area under the curve as the primary optimization metric when dealing with imbalanced datasets. Since many of the datasets were imbalanced, area under the curve was chosen as the primary optimization metric. Each job was also set for a maximum training time of 24 hours. AutoML automatically enables featurization by default for all jobs involving nonnumeric datasets.

3.5 Model Evaluation

This section describes the approach used to evaluate the AutoML-generated models. Each of the generated models was tested using test datasets to assess their ability to correctly detect vulnerable smart contracts. Testing using each row in the test dataset

produced a true positive (TP), true negative (TN), false positive (FP), or false negative (FN) result. Using these values, performance evaluation metrics such as accuracy, precision, recall, F1 score, and Matthews correlation coefficient can be calculated. In addition, the duration metrics associated with training and testing using GPU and CPU-based clusters were captured to assess any differences in performance between compute cluster types.

3.5.1 Confusion Matrix

A confusion matrix provides a visual representation of how well a model predicts. With binary classification models, a confusion matrix can present four possible outputs (Tharwat, 2018): TP, TN, FP, and FN. A TP occurs when the model correctly predicts the presence of a vulnerability. When the model correctly predicts the absence of a vulnerability, it is considered a TN. A FP occurs when the model incorrectly predicts a vulnerability when there is no actual vulnerability present. A FP is also often referred to as a Type I error. A FN occurs when the model incorrectly predicts the absence of a vulnerability when there is an actual vulnerability present. A FN is also called a Type II error. A confusion matrix visually presents these values. Figure 7 shows the format of a confusion matrix.

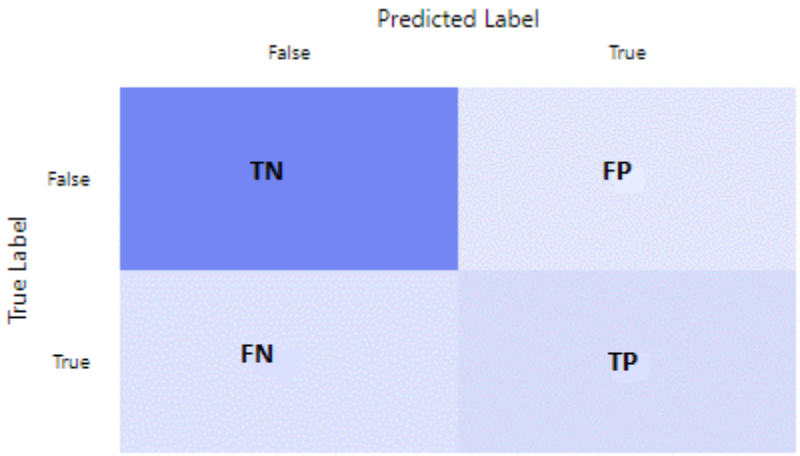

Figure 7—Confusion Matrix Format

3.5.2 Accuracy

The accuracy metric is the total number of correctly predicted values divided by the total number of values tested (Hossin & Sulaiman, 2015). Accuracy represents the degree to which predicted values are close to true values. The formula to calculate accuracy is as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

where,

TP = number of true positives,

TN = number of true negatives,

FP = number of false positives, and

FN = number of false negatives.

Equation 1—Formula for Accuracy

3.5.3 Precision

The precision metric is the number of correctly identified positive values divided by the total number of positive predictions. Precision represents the degree to which the predicted values are close to each other. The formula to calculate precision is as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

where,

TP = number of true positives, and

FP = number of false positives,

Equation 2—Formula for Precision

3.5.4 Recall

The recall metric is the number of correctly identified positive values divided by the total number of correctly identified positive and incorrectly identified positive values. Recall is sometimes referred to as the true positive rate. The formula to calculate recall is as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

where,

TP = number of true positives, and

FN = number of false negatives.

Equation 3—Formula for Recall

3.5.5 F1 Score

The F1 score is the harmonic mean between the recall and precision metrics (Hossin & Sulaiman, 2015). The formula to calculate the F1 score is as follows:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Equation 4—Formula for F1 Score

3.5.6 Matthews Correlation Coefficient

The Matthews correlation coefficient is a useful metric when dealing with imbalanced datasets. The metric ranges between −1 and +1. A score of +1 is generated with perfect models, 0 correlates with a random guess, and −1 is generated for poor models. This metric only produces a high positive value when the model can correctly predict the majority of both negative and positive values (Chicco & Jurman, 2020). The formula to calculate the Matthews correlation coefficient is as follows:

$$\mathrm{MCC} = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}}$$

where,

TP = number of true positives,

TN = number of true negatives,

FP = number of false positives, and

FN = number of false negatives.

Equation 5—Formula for Matthews Correlation Coefficient

3.5.7 Duration

The length of time taken to complete model selection and training was measured in minutes for both GPU- and CPU-based experiments. The amount of time taken to test the models using the respective compute clusters was also calculated to determine whether there is a timing benefit to using GPUs over CPUs. The amount of time taken by the built vulnerability detection models to evaluate smart contracts was measured in seconds.

Chapter 4—Results

4.1 Introduction

Section 4.2 presents the findings from the exploratory data analysis (Section 3.2.2). Section 4.3 describes the outcomes from the data preparation step. Sections 4.4 through 4.9 present the results of the experiments carried out to test the six hypotheses. In each of the experiments, models were selected and trained using AutoML, using both graphics processing unit (GPU) and central processing unit (CPU)-based compute clusters. A confusion matrix along with performance metrics such as accuracy, precision, recall, F1 score, Matthews correlation coefficient (MCC), and duration was calculated for the experiments. Section 4.10 provides an overall summary of the results.

4.2 Exploratory Data Analysis

There were 12 distinct data fields in the downloaded smart contract source code dataset. All the fields were textual in nature. As a result, many of the descriptive statistics are not available for this dataset. The available summary statistics on the 12 fields of the dataset are presented in Figure 8. Most of the fields provide metadata on the smart contracts that were collected by Etherscan. Given that the focus of this research is to study the effectiveness of identifying smart contract vulnerabilities based on smart contract source code prior to deployment to the Ethernet blockchain network, it is prudent to focus efforts on collecting source code data. A decision was therefore made to exclude 10 of the 12 fields from the dataset, leaving just the smart contract address and source code fields. The address field is not necessary for model training purposes; however, it is required as a key for correlating source code with the appropriate vulnerability data from the ScrawlD dataset and ultimately correlating results with the appropriate smart contract.

AutoML treats source code data as text and uses a variety of natural-language processing techniques to generate additional features.

Figure 8—Smart Contract Dataset Summary Statistics

To further explore the data on source code features, a word cloud was generated using Exploratory Desktop. The word cloud revealed that there were 149,429 unique words and 717,448 unique word pairs in the source code for the 9,252 smart contracts. Tokenization of the words yielded 7,223,550 unique tokens. This demonstrates that while technically only one field was used, the large amount of text involved should allow for ample unique patterns to form to help identify source code that may be prone to vulnerabilities. The top 300 words are shown as a word cloud in Figure 9. The top 30

word pairs are shown in Figure 10.

Figure 9—Word Cloud of Top 300 Words in Smart Contract Source Code Dataset

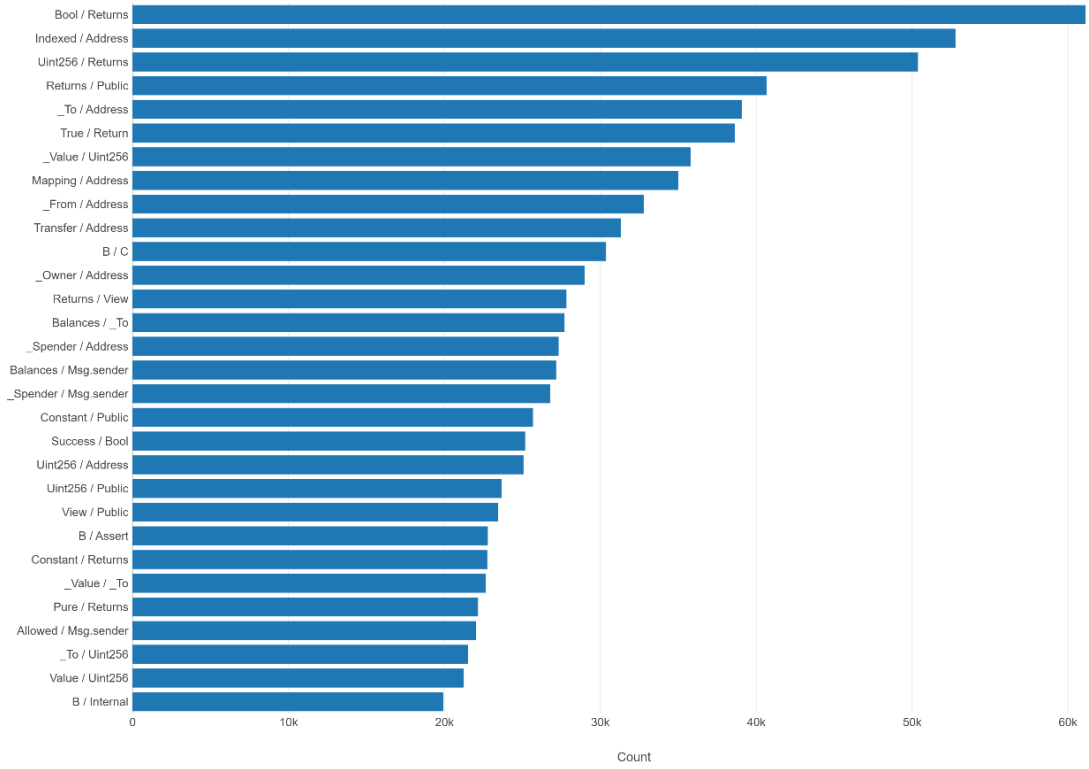

Figure 10—Top 30 Word Pairs in Smart Contract Source Code Dataset

Each row in the file lists the contract address along with a listing of known vulnerabilities with that contract. Figure 11 shows the distribution of the various vulnerability combinations among this group of vulnerable contracts. The pie chart in Figure 12 illustrates the percentage of smart contracts with vulnerabilities in the overall dataset. Figure 13 represents the general distribution of vulnerability types in the dataset.

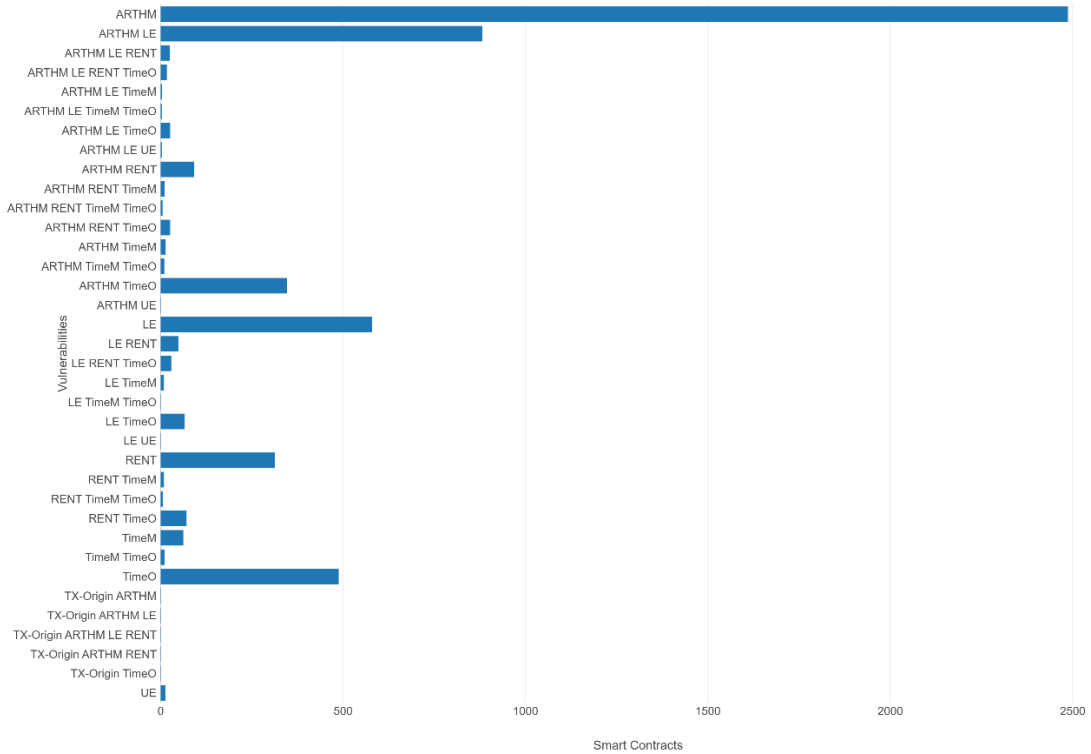

Figure 11—Vulnerability Sets in the ScrawlD Dataset

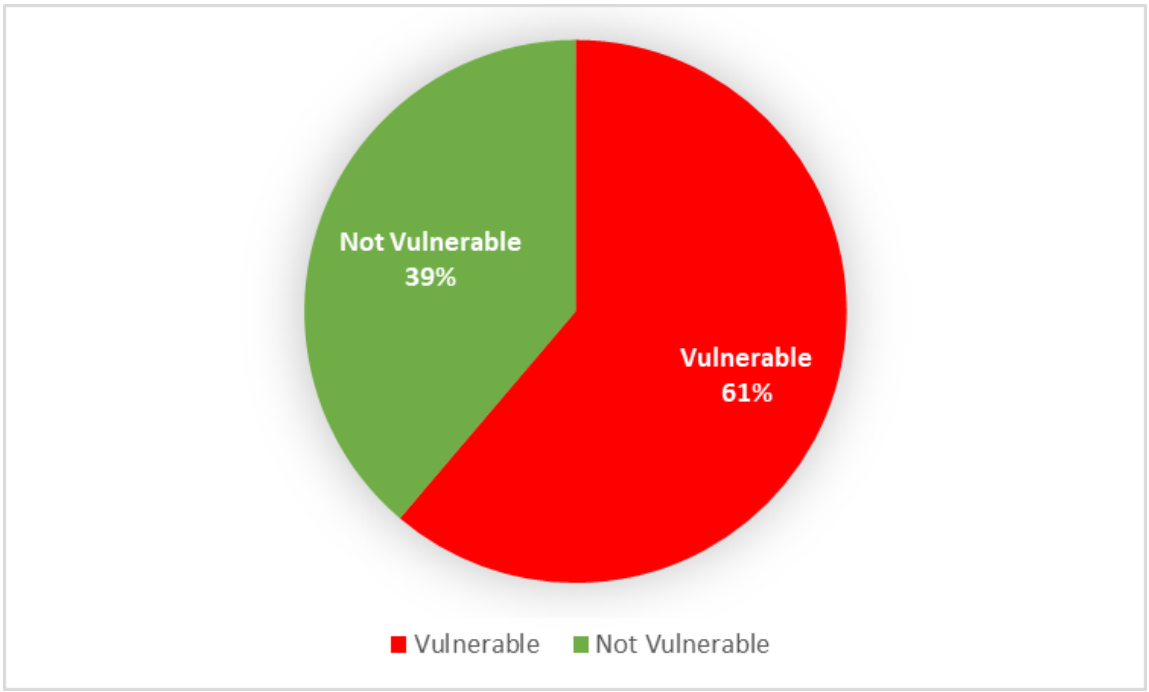

Figure 12—Percentage of Vulnerable Smart Contracts in the ScrawlD Dataset

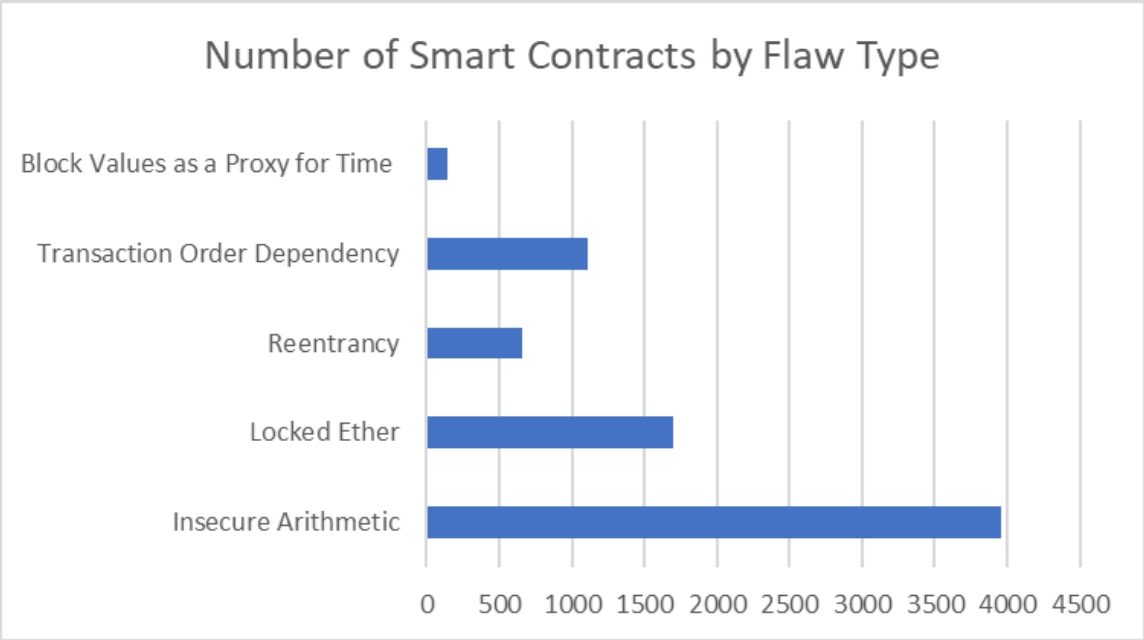

Figure 13—Distribution of Vulnerabilities in the ScrawlD Dataset

4.3 Data Preparation

Following the methodology described in Section 3.2.4, training and test datasets were created from the master dataset. Table 3 and Figure 14 show the distribution of the various vulnerabilities within the training datasets, and Table 4 and Figure 15 show the distribution of the various vulnerabilities within the test datasets. Table 15 in Appendix A shows a mapping between the ScrawlD dataset vulnerability labels and the datasets prepared during this praxis research that contain the corresponding vulnerability type.

Training Dataset	No. of Smart Contracts With Vulnerabilities	No. of Smart Contracts Without Vulnerabilities
VulnerableOrNot	4,581	2,821
Insecure Arithmetic Flaws	3,208	4,194
Block Values as a Proxy for Time Flaws	125	7,277
Reentrancy Flaws	576	6,826
Locked Ether Flaws	1352	6,050
Transaction Order Dependency Flaws	859	6,543

Table 3—Training Datasets Overview

Test Dataset	No. of Smart Contracts With Vulnerabilities	No. of Smart Contracts Without Vulnerabilities
VulnerableOrNot	1,083	767
Insecure Arithmetic Flaws	748	1,102
Block Values as a Proxy for Time Flaws	20	1,830
Reentrancy Flaws	78	1,772
Locked Ether Flaws	344	1,506
Transaction Order Dependency Flaws	247	1,603

Table 4—Test Datasets Overview

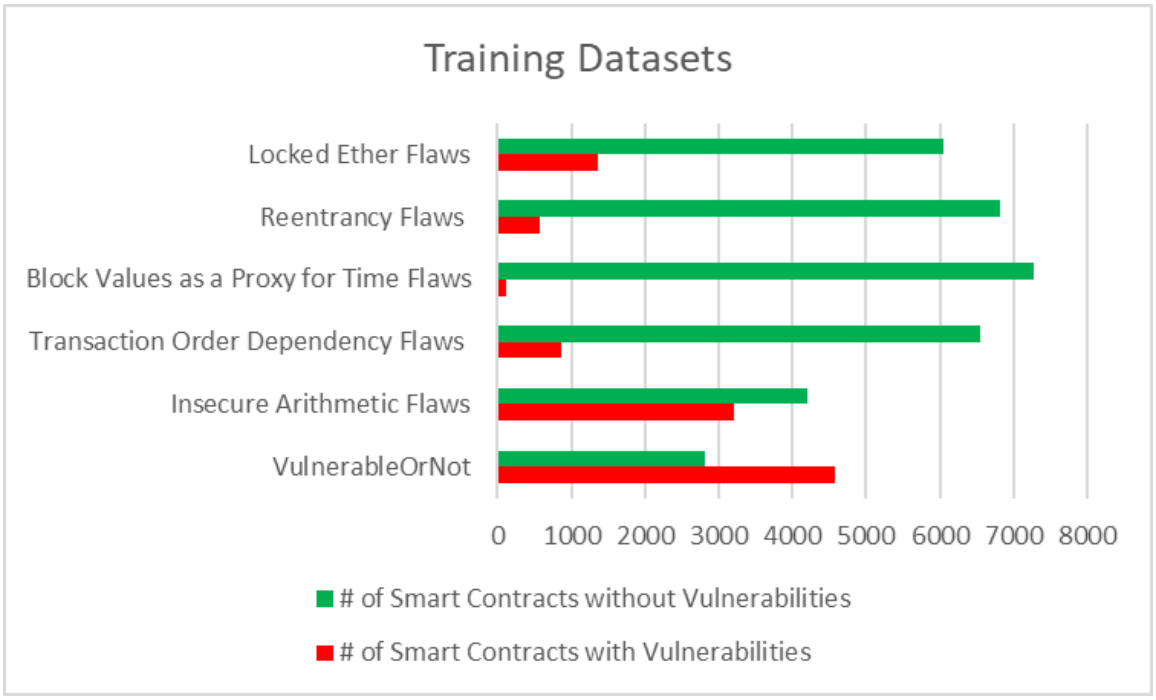

Figure 14—Distribution of Vulnerabilities in Training Datasets

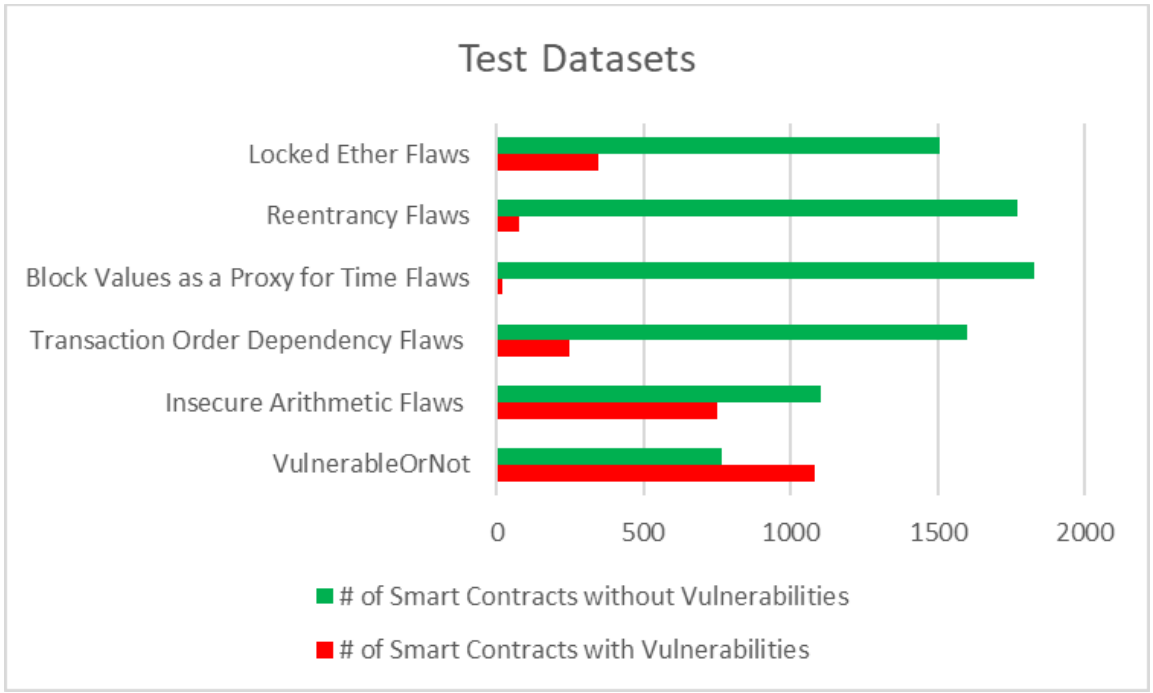

Figure 15—Distribution of Vulnerabilities in Test Datasets

4.4 Hypothesis 1 Results

This section presents the results of the experiments testing Hypothesis 1, which postulates that cloud-based automated machine learning can be used to build a model that identifies smart contracts with at least one of the common types of vulnerabilities with accuracy greater than 80% and precision greater than 80%.

The CPU-based AutoML experiment took 5 hours and 8 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Thirty-one classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This LightGBM-based model was tested using the VulnerableOrNot test dataset. During testing, this model achieved an accuracy of 85.18%, precision of 84.60%, recall of 91.32%, F1 score of 87.83%, and a Matthews correlation coefficient of 0.6931. Testing 1,850 smart contracts took 65 seconds. On average, each smart contract took 0.0351 seconds to test.

The GPU-based AutoML experiment took 3 hours and 4 minutes. During the featurization stage, 850,601 engineered features were generated to support model training. Fifty-four classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and StandardScalerWrapper standardization algorithm was determined to be the best-performing model. This LightGBM-based model was tested using the VulnerableOrNot test dataset. During testing, this model achieved an accuracy of 85.35%, precision of 84.46%, recall of 91.87%, F1 score of 88.01%, and a Matthews correlation coefficient of 0.6967. Testing

1,850 smart contracts took 71 seconds. On average, each smart contract took 0.0384 seconds to test.

The test results for both of these leading models using the VulnerableOrNot test dataset are summarized in Table 5. Figure 16 shows the confusion matrix for the CPU-based general vulnerability flaws detection model, and Figure 17 shows the confusion matrix for the GPU-based general vulnerability flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.8518	0.8535
Precision	0.8460	0.8446
Recall	0.9132	0.9187
F1 Score	0.8783	0.8801
MCC	0.6931	0.6967
Model testing duration (s)	65	71
Average time to test a single contract (s)	0.0351	0.0384

Table 5—General Vulnerability Detection Model Test Results

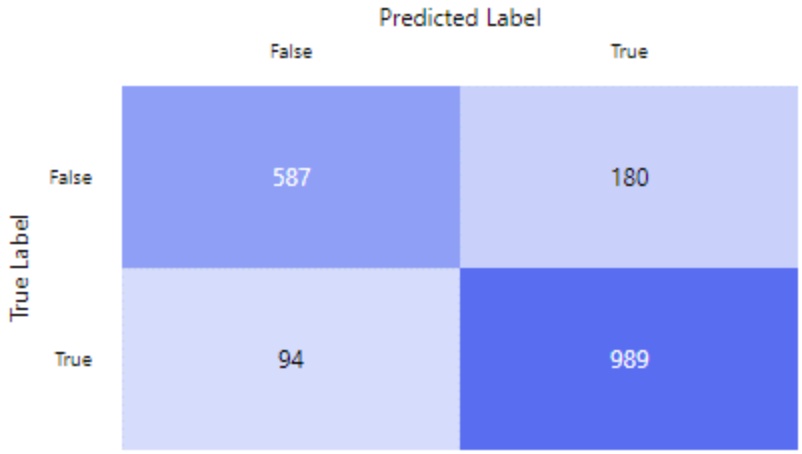

Figure 16—Confusion Matrix for CPU-Based General Vulnerability Detection Model

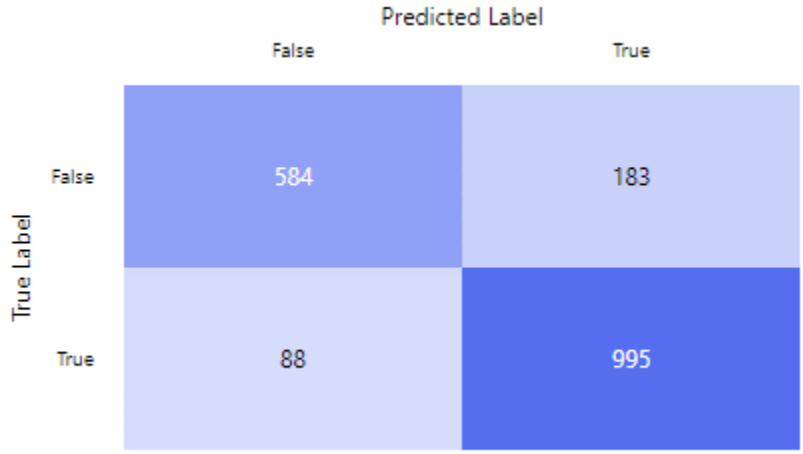

Figure 17—Confusion Matrix for GPU-Based General Vulnerability Detection Model

4.5 Hypothesis 2 Results

This section presents the results of the experiments testing Hypothesis 2, which states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as insecure arithmetic flaws with accuracy greater than 80% and precision greater than 80%.

The CPU-based AutoML experiment took 5 hours and 35 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Thirty classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This LightGBM-based model was tested using the Insecure Arithmetic Flaws test dataset. During testing, this model achieved an accuracy of 86.81%, precision of 85.79%, recall of 80.74%, F1 score of 83.19%, and a Matthews correlation coefficient of 0.7244. Testing 1,850 smart contracts took 69 seconds. On average, each smart contract took 0.0371 seconds to test.

The GPU-based AutoML experiment took 3 hours and 20 minutes. During the featurization stage, 857,794 engineered features were generated to support model training. Forty-nine classification models were trained and evaluated by AutoML for this experiment. A model using the XGBoostClassifier classifier and SparseNormalizer standardization algorithm was determined to be the best-performing model. This XGBoostClassifier-based model was tested using the Insecure Arithmetic Flaws test dataset. During testing, this model achieved an accuracy of 87.45%, precision of 87.17%, recall of 80.88%, F1 score of 83.91%, and a Matthews correlation coefficient of 0.7379. Testing 1,850 smart contracts took 58 seconds. On average, each smart contract took 0.0314 seconds to test.

The test results for both leading models using the Insecure Arithmetic Flaws test dataset are summarized in Table 6. Figure 18 shows the confusion matrix for the CPU-based insecure arithmetic flaws detection model, and Figure 19 shows the confusion matrix for the GPU-based insecure arithmetic flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.8681	0.8745
Precision	0.8579	0.8717
Recall	0.8074	0.8088
F1 Score	0.8319	0.8391
MCC	0.7244	0.7379
Model testing duration (s)	69	58
Average time to test a single contract (s)	0.0373	0.0314

Table 6—Insecure Arithmetic Flaws Detection Model Test Results

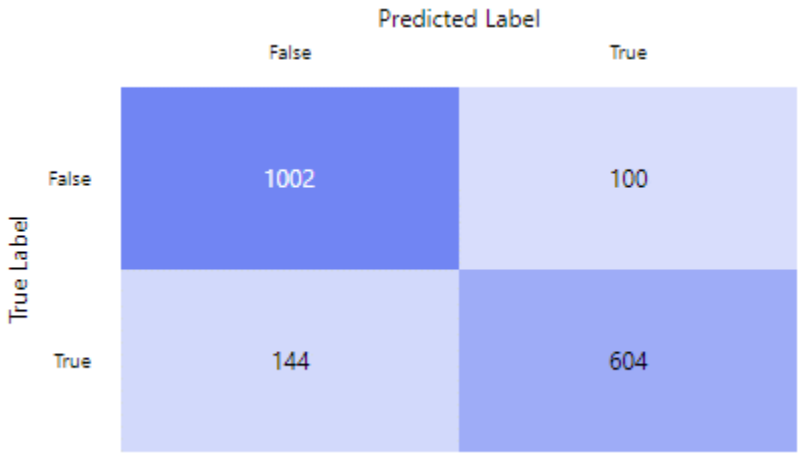

Figure 18—Confusion Matrix for CPU-Based Insecure Arithmetic Flaws Detection Model

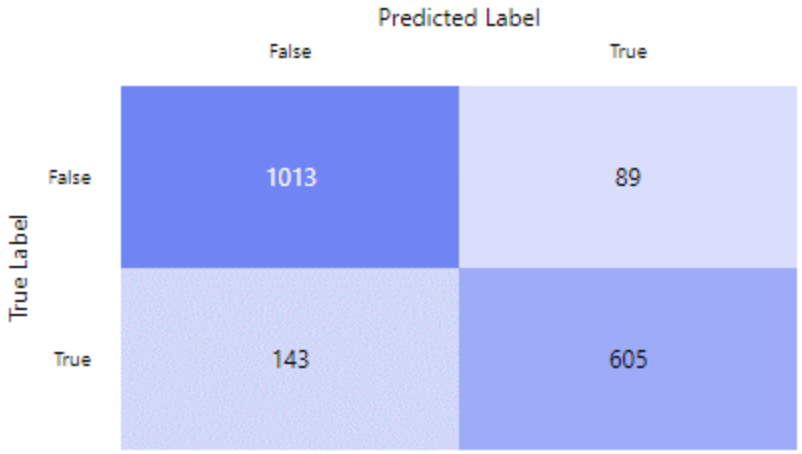

Figure 19—Confusion Matrix for GPU-Based Insecure Arithmetic Flaws Detection Model

4.6 Hypothesis 3 Results

This section presents the results of the experiments testing Hypothesis 3, which states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as block values as a proxy for time flaws with accuracy greater than 80% and precision greater than 80%.

The compute duration for the CPU-based AutoML experiment was 6 hours and 15 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Nineteen classification models were trained and evaluated by AutoML for this experiment. A model using the VotingEnsemble classifier was determined to be the best-performing model. This voting ensemble classifier-based model was tested using the Block Values as a Proxy for Time Flaws test dataset. During testing, this model achieved an accuracy of 98.86%, precision of 46.15%, recall of 30%, F1 score of 36.36%, and a Matthews correlation coefficient of 0.3666. Testing 1,850 smart contracts took 67 seconds. On average, each smart contract took 0.0362 seconds to test.

The compute duration for the GPU-based AutoML experiment was 10 hours and 25 minutes. During the featurization stage, 849,265 engineered features were generated to support model training. Forty classification models were trained and evaluated by AutoML for this experiment. A model using the ExtremeRandomTrees classifier and StandardScalerWrapper standardization algorithm was determined to be the best-performing model. This ExtremeRandomTrees classifier-based model was tested using the Block Values as a Proxy for Time Flaws test dataset. During testing, this model achieved an accuracy of 88.59%, precision of 6.78%, recall of 75%, F1 score of 12.44%, and a Matthews correlation coefficient of 0.2032. Testing 1,850 smart contracts took 57 seconds. On average, each smart contract took 0.0308 seconds to test.

The test results for both leading models using the Block Values as a Proxy for Time Flaws test dataset are summarized in Table 7. Figure 20 shows the confusion matrix for the CPU-based block values as a proxy for time flaws detection model, and Figure 21 shows the confusion matrix for the GPU-based block values as a proxy for time flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.9886	0.8859
Precision	0.4615	0.0678
Recall	0.3	0.75
F1 Score	0.3636	0.1244
MCC	0.3666	0.2032
Model testing duration (s)	67	57
Average time to test a single contract (s)	0.0362	0.0308

Table 7—Block Values as a Proxy for Time Flaws Detection Model Test Results

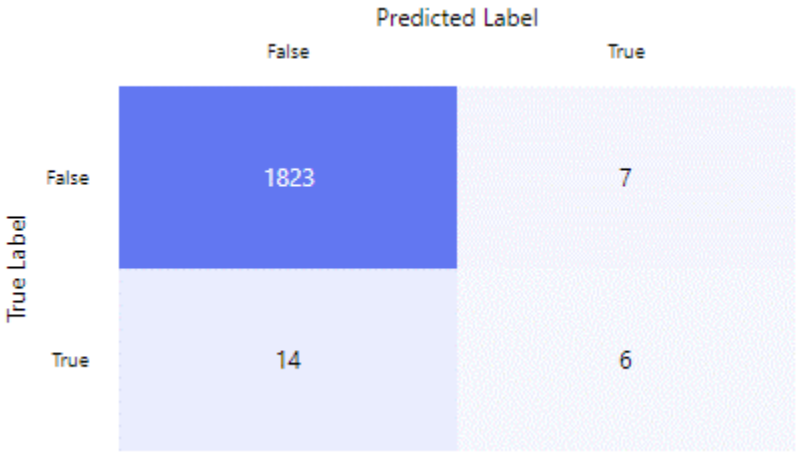

Figure 20—Confusion Matrix for CPU-Based Block Values as a Proxy for Time Flaws Detection Model

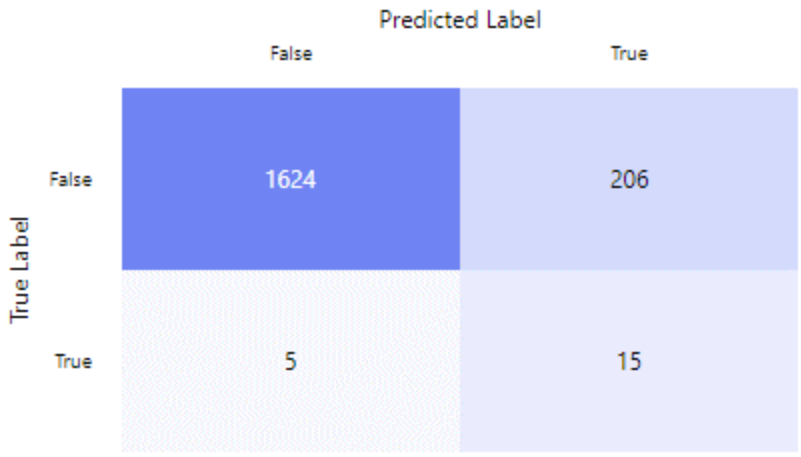

Figure 21—Confusion Matrix for GPU-Based Block Values as a Proxy for Time Flaws Detection Model

4.7 Hypothesis 4 Results

This section covers the results of the experiments testing Hypothesis 4, which states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as reentrancy flaws with accuracy greater than 80% and precision greater than 80%.

The compute duration for the CPU-based AutoML experiment was 6 hours and 15 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Twenty-nine classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and MaxAbsScaler standardization algorithm was determined to be the best performing. This LightGBM classifier-based model was tested using the Reentrancy Flaws test dataset. During testing, this model achieved an accuracy of 94.97%, precision of 41.17%, recall of 44.87%, F1 score of 42.94%, and a Matthews correlation coefficient of 0.4036. Testing 1,850 smart contracts took 59 seconds. On average, each smart contract took 0.0319 seconds to test.

The compute duration for the GPU-based AutoML experiment was 10 hours and 25 minutes. During the featurization stage, 850,933 engineered features were generated to support model training. Ninety-eight classification models were trained and evaluated by AutoML for this experiment. A model using the XGBoostClassifier classifier and SparseNormalizer standardization algorithm was determined to be the best-performing model. This XGBoostClassifier-based model was tested using the Reentrancy Flaws test dataset. During testing, this model achieved an accuracy of 95.35%, precision of 43.93%, recall of 37.17%, F1 score of 40.27%, and a Matthews correlation coefficient of 0.3801. Testing 1,850 smart contracts took 57 seconds. On average, each smart contract took 0.0308 seconds to test.

The test results for both leading models using the Reentrancy Flaws test dataset are summarized in Table 8. Figure 22 shows the confusion matrix for the CPU-based reentrancy flaws detection model, and Figure 23 shows the confusion matrix for the GPU-based reentrancy flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.9497	0.9535
Precision	0.4117	0.4393
Recall	0.4487	0.3717
F1 Score	0.4294	0.4027
MCC	0.4036	0.3801
Model testing duration (s)	59	57
Average time to test a single contract (s)	0.0319	0.0308

Table 8—Reentrancy Flaws Detection Model Test Results

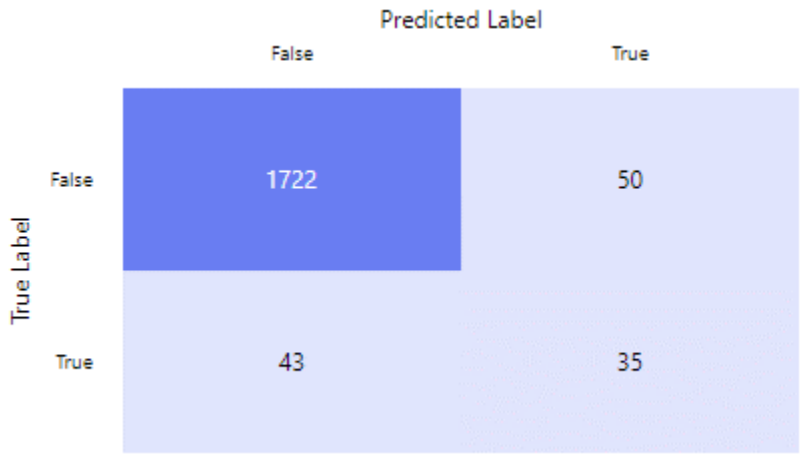

Figure 22—Confusion Matrix for CPU-Based Reentrancy Flaws Detection Model

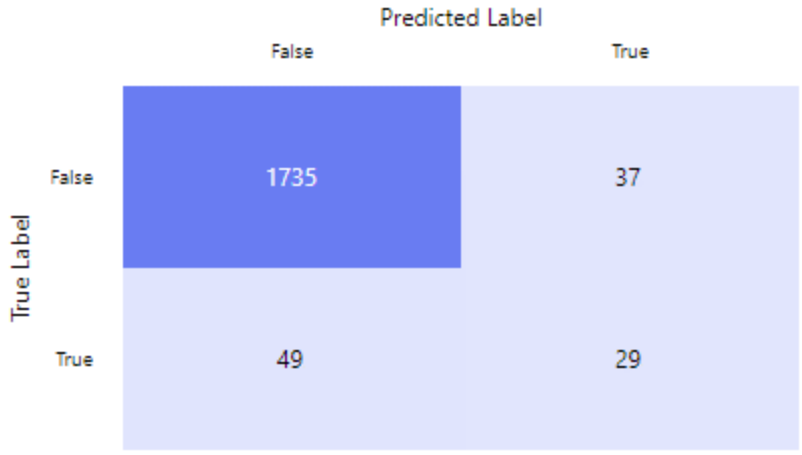

Figure 23—Confusion Matrix for GPU-Based Reentrancy Flaws Detection Model

4.8 Hypothesis 5 Results

This section presents the results of the experiments testing Hypothesis 5, which states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as locked Ether flaws with accuracy greater than 80% and precision greater than 80%.

The CPU-based AutoML experiment took 5 hours and 25 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Thirty-one classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This LightGBM classifier-based model was tested using the Locked Ether Flaws test dataset. During testing, this model achieved an accuracy of 96.27%, precision of 95.68%, recall of 83.72%, F1 score of 89.30%, and a Matthews correlation coefficient of 0.8734. Testing 1,850 smart contracts took 68 seconds. On average, each smart contract took 0.0368 seconds to test.

The GPU-based AutoML experiment took 2 hours and 28 minutes. During the featurization stage, 856,928 engineered features were generated to support model

training. Forty classification models were trained and evaluated by AutoML for this experiment. A model using the XGBoostClassifier classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This XGBoostClassifier-based model was tested using the Locked Ether Flaws test dataset. During testing, this model achieved an accuracy of 95.78%, precision of 95.54%, recall of 81.10%, F1 score of 87.73%, and a Matthews correlation coefficient of 0.8562. Testing 1,850 smart contracts took 57 seconds. On average, each smart contract took 0.0308 seconds to test.

The test results for both leading models using the Locked Ether Flaws test dataset are summarized in Table 9. Figure 24 shows the confusion matrix for the CPU-based locked Ether flaws detection model, and Figure 25 shows the confusion matrix for the GPU-based locked Ether flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.9627	0.9578
Precision	0.9568	0.9554
Recall	0.8372	0.8110
F1 Score	0.8930	0.8773
MCC	0.8734	0.8562
Model testing duration (s)	68	57
Average time to test a single contract (s)	0.0368	0.0308

Table 9—Locked Ether Flaws Detection Model Test Results

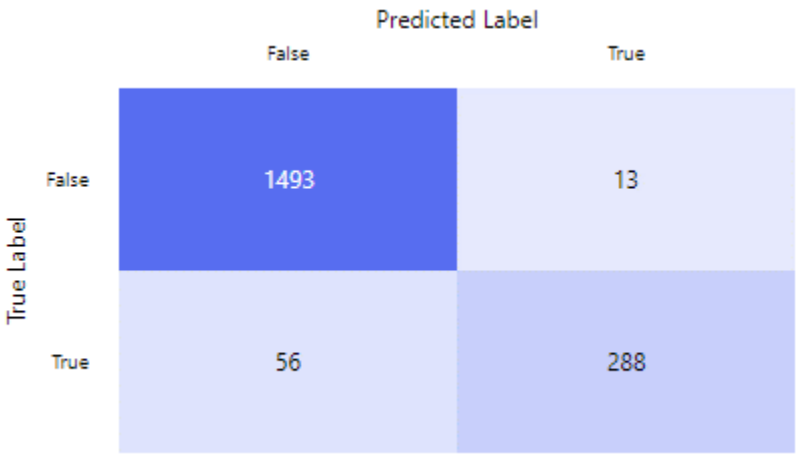

Figure 24—Confusion Matrix for CPU-Based Locked Ether Flaws Detection Model

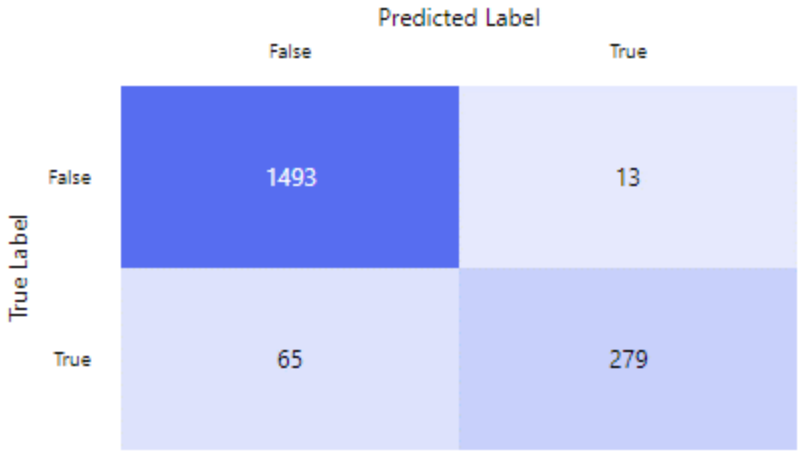

Figure 25—Confusion Matrix for GPU-Based Locked Ether Flaws Detection Model

4.9 Hypothesis 6 Results

This section presents the results of the experiments testing Hypothesis 6, which states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as transaction order dependency flaws with accuracy greater than 80% and precision greater than 80%.

The CPU-based AutoML experiment took 4 hours and 10 minutes. During the featurization stage, 912,371 engineered features were generated to support model training. Thirty-one classification models were trained and evaluated by AutoML for this experiment. A model using the LightGBM classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This LightGBM classifier-based model was tested using the Transaction Order Dependency Flaws test dataset. During testing, this model achieved an accuracy of 94.86%, precision of 88.77%, recall of 70.44%, F1 score of 78.55%, and a Matthews correlation coefficient of 0.7633. Testing 1,850 smart contracts took 61 seconds. On average, each smart contract took 0.0330 seconds to test.

The GPU-based AutoML experiment took 2 hours and 39 minutes. During the featurization stage, 851,080 engineered features were generated to support model training. Forty classification models were trained and evaluated by AutoML for this experiment. A model using the XGBoostClassifier classifier and MaxAbsScaler standardization algorithm was determined to be the best-performing model. This LightGBM classifier-based model was tested using the Transaction Order Dependency Flaws test dataset. During testing, this model achieved an accuracy of 95.01%, precision of 90.15%, recall of 70.44%, F1 score of 79.09%, and a Matthews correlation coefficient of 0.7706. Testing 1,850 smart contracts took 57 seconds. On average, each smart contract took 0.0308 seconds to test.

The test results for both leading models using the Transaction Order Dependency Flaws test dataset are summarized in Table 10. Figure 26 shows the confusion matrix for the CPU-based transaction order dependency flaws detection model, and Figure 27 shows the confusion matrix for the GPU-based transaction order dependency flaws detection model.

	Compute Cluster Type	
	CPU	**GPU**
Accuracy	0.9486	0.9502
Precision	0.8877	0.9015
Recall	0.7044	0.7044
F1 Score	0.7855	0.7909
MCC	0.7633	0.7706
Model testing duration (s)	61	57
Average time to test a single contract (s)	0.0330	0.0308

Table 10—Transaction Order Dependency Flaws Detection Model Test Results

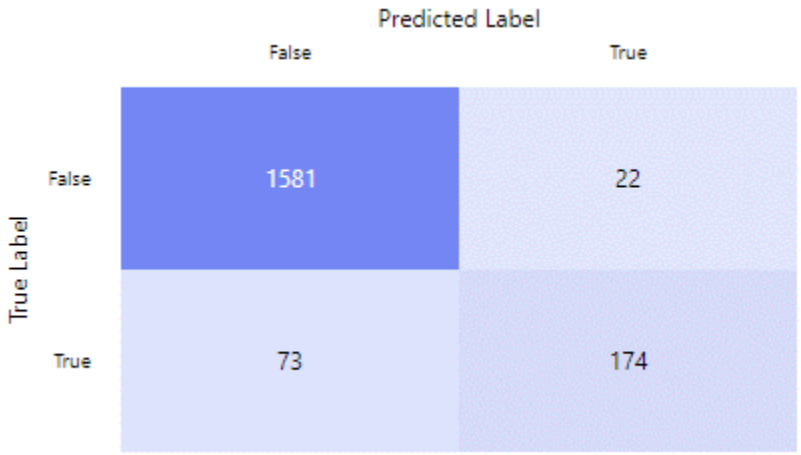

Figure 26—Confusion Matrix for CPU-Based Transaction Order Dependency Flaws Detection Model

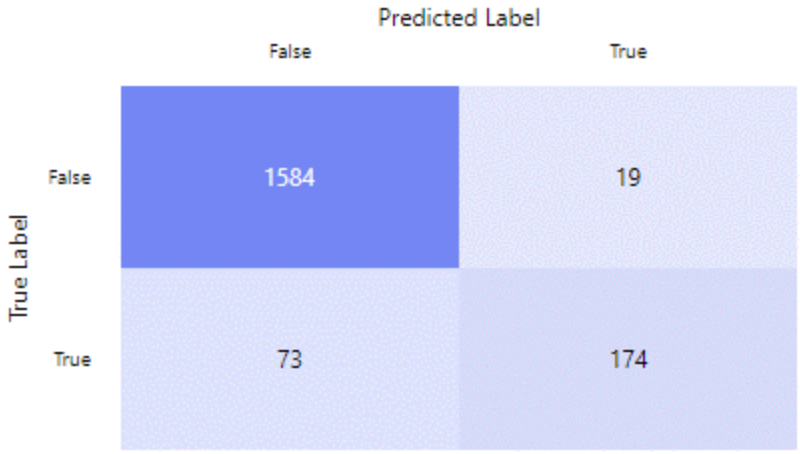

Figure 27—Confusion Matrix for GPU-Based Transaction Order Dependency Flaws Detection Model

4.10 Results Summary

The overall statistical results from the experiments are summarized in Tables 11 and 12 and visually depicted in Figures 28 and 29. Duration results for CPU and GPU experiments are summarized in Table 13 and visually depicted in Figures 30 and 31.

AutoML Model	Accuracy	Precision	Recall	F1 Score	MCC
VulnerableOrNot	85.18%	84.60%	91.32%	87.83%	0.6931
Insecure Arithmetic Flaws	86.81%	85.79%	80.74%	83.19%	0.7244
Block Values as a Proxy for Time Flaws	98.86%	46.15%	30.00%	36.36%	0.3666
Reentrancy Flaws	94.97%	41.17%	44.87%	42.94%	0.4036
Locked Ether Flaws	96.27%	95.68%	83.72%	89.30%	0.8734
Transaction Order Dependency Flaws	94.86%	88.77%	70.44%	78.55%	0.7633

Table 11—Overall Test Results for CPU-Based Vulnerability Detection Models

AutoML Model	Accuracy	Precision	Recall	F1 Score	MCC
VulnerableOrNot	85.35%	84.46%	91.87%	88.01%	0.6967
Insecure Arithmetic Flaws	87.45%	87.17%	80.88%	83.91%	0.7379
Block Values as a Proxy for Time Flaws	88.59%	6.78%	75.00%	12.44%	0.2032
Reentrancy Flaws	95.35%	43.93%	37.17%	40.27%	0.3801
Locked Ether Flaws	95.78%	95.54%	81.10%	87.73%	0.8562
Transaction Order Dependency Flaws	95.01%	90.15%	70.44%	79.09%	0.7706

Table 12—Overall Test Results for GPU-Based Vulnerability Detection Models

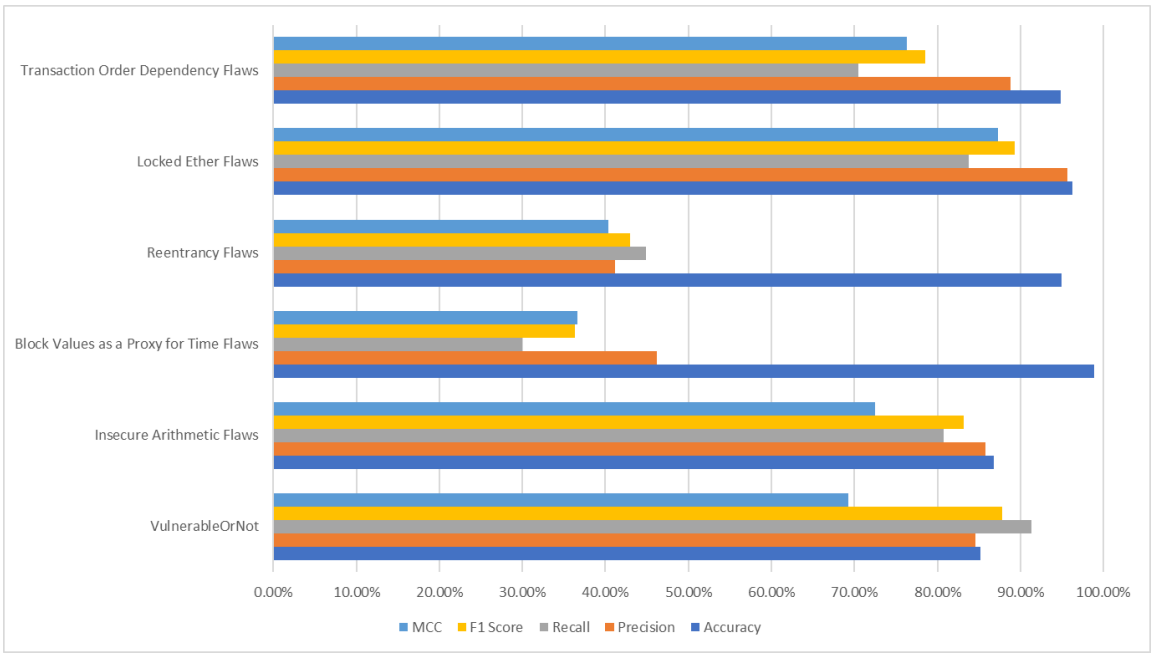

Figure 28—Overall Test Results for CPU-Based Vulnerability Detection Models

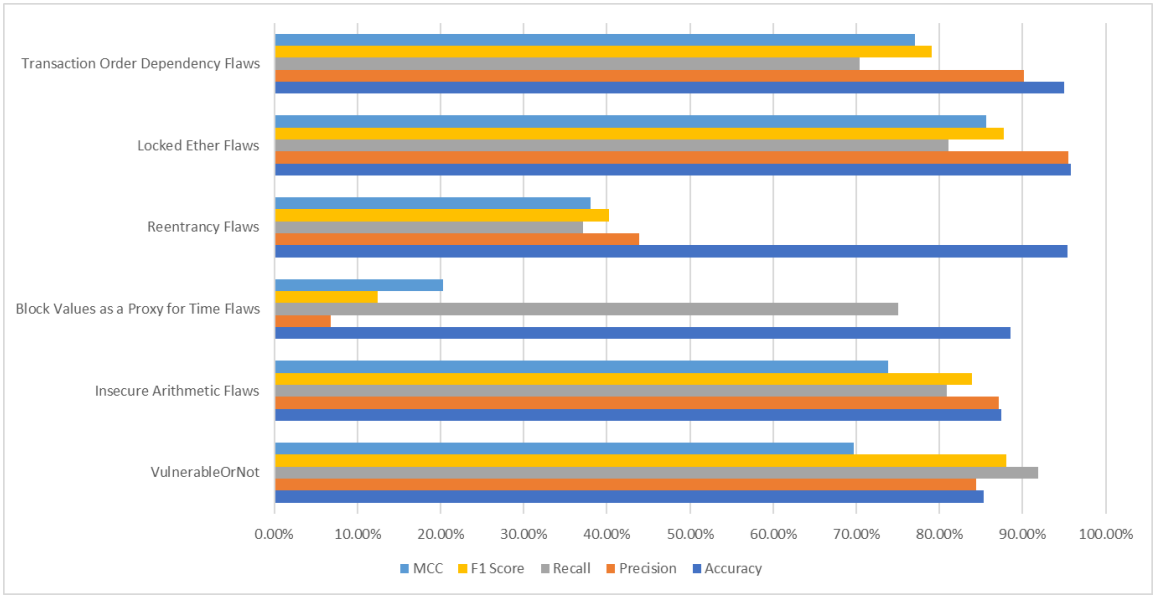

Figure 29—Overall Test Results for GPU-Based Vulnerability Detection Models

AutoML Model	Training Time for CPU-Based Models (min)	Training Time for GPU-Based Models (min)	Testing Time for CPU-Based Models (s)	Testing Time for GPU-Based Models (s)
VulnerableOrNot	308	184	0.0351	0.0384
Insecure Arithmetic Flaws	335	200	0.0373	0.0314
Block Values as a Proxy for Time Flaws	1474	225	0.0362	0.0308
Reentrancy Flaws	375	625	0.0319	0.0308
Locked Ether Flaws	325	148	0.0368	0.0308
Transaction Order Dependency Flaws	250	159	0.0330	0.0308

Table 13—Training and Testing Duration for GPU and CPU-Based Models

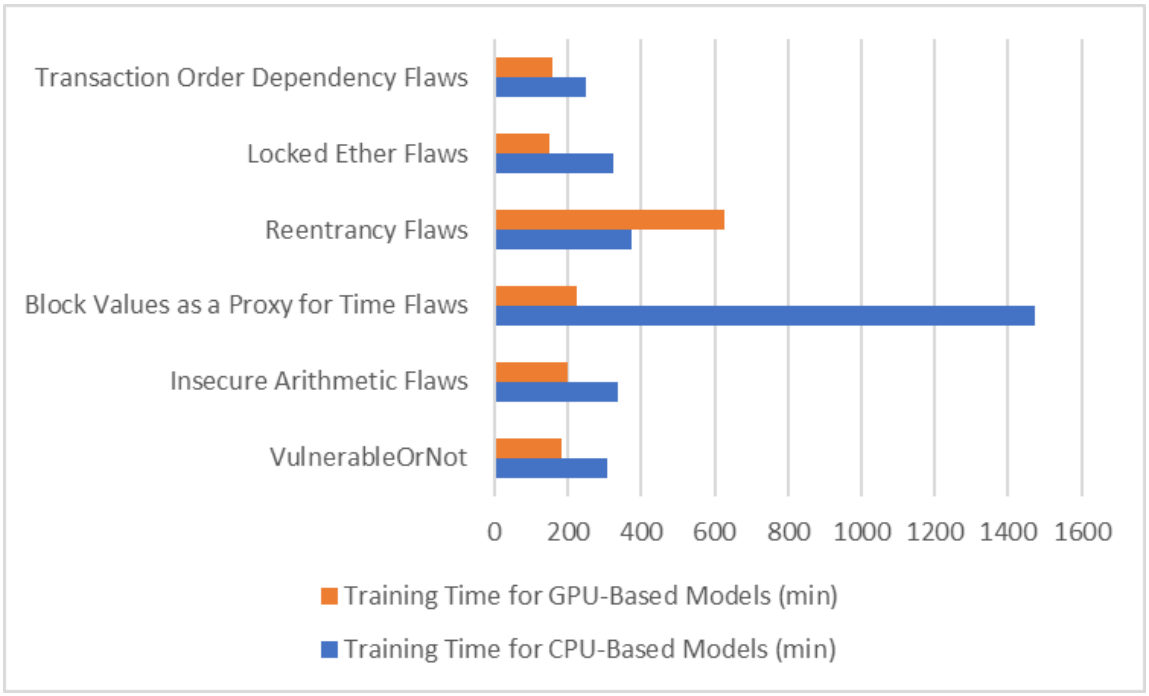

Figure 30—Model Training Time

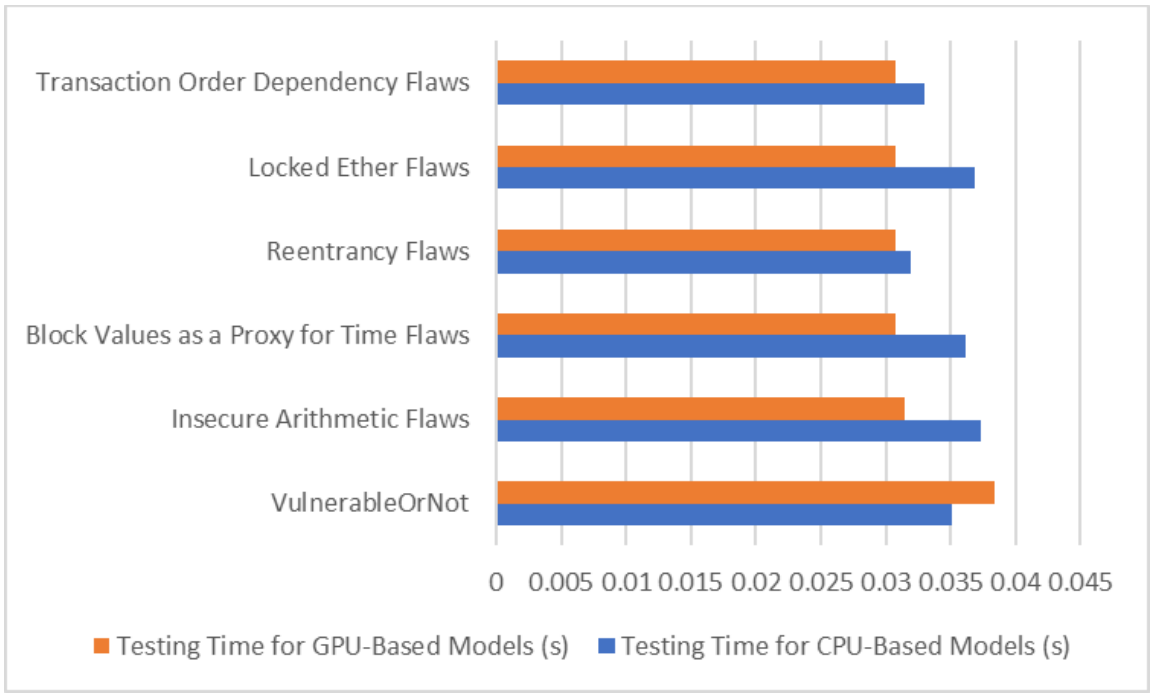

Figure 31—Average per Smart Contract Testing Time

Chapter 5—Discussion and Conclusions

5.1 Discussion

The objective of this praxis was to study the effectiveness of using cloud-based automated machine learning (AutoML) capabilities to develop models that can correctly identify smart contracts with common vulnerabilities. Four research questions and six hypotheses were used to guide the study. Experiments were designed and conducted to collect the data needed to answer the research questions and verify the hypotheses. In this chapter, the results of the experiments are reviewed to verify the hypotheses and address the research questions.

5.1.1 Verification of Hypotheses

Hypothesis 1 states that cloud-based AutoML can be used to build a model that detects vulnerable smart contracts with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.4, both central processing unit (CPU) and graphical processing unit (GPU)-based testing resulted in accuracy and precision scores greater than 80%. These results therefore verify Hypothesis 1. This verification is supported by acceptable metrics for recall, F1 score, and MCC on this model.

Hypothesis 2 states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as insecure arithmetic flaws with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.5, both CPU- and GPU-based testing resulted in accuracy and precision scores greater than 80%. These results therefore verify Hypothesis 2. This

verification is supported by acceptable metrics for recall, F1 score, and MCC on this model.

Hypothesis 3 states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as block values as a proxy for time flaws with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.6, while both CPU- and GPU-based testing resulted in accuracy greater than 80%, the precision scores were well below the hypothesis threshold. The results therefore do not verify Hypothesis 3. One reason AutoML could not build an effective model may be that the training dataset was highly imbalanced and only contained 125 examples of smart contracts with the block values as a proxy for time flaw. The high accuracy score can be attributed to the class imbalance. As shown in the results, however, the MCC and F1 scores were low for this model, which is typical for a model trained using an imbalanced dataset.

Hypothesis 4 states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as reentrancy flaws with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.7, while both CPU- and GPU-based testing resulted in accuracy greater than 80%, the precision scores were well below the hypothesis target. The results therefore do not verify Hypothesis 4. One reason AutoML could not build an effective model may be that the training dataset was highly imbalanced and only contained 576 examples of smart contracts with the reentrancy flaw. The high accuracy score can be attributed to the class imbalance. As shown in the results, however, the recall, MCC, and F1 scores were low for this model, which is typical for a model trained using an imbalanced dataset.

Hypothesis 5 states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as locked Ether flaws with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.8, both CPU- and GPU-based testing resulted in accuracy and precision scores greater than 80%. The results therefore verify Hypothesis 5. This verification is supported by acceptable metrics for recall, F1 score, and MCC on this model.

Hypothesis 6 states that cloud-based AutoML can be used to build a model that identifies smart contracts with vulnerabilities classified as transaction order dependency flaws with accuracy greater than 80% and precision greater than 80%. According to the results presented in Section 4.9, both CPU- and GPU-based testing resulted in accuracy and precision scores greater than 80%. The results therefore verify Hypothesis 6. This verification is supported by acceptable metrics for recall, F1 score, and MCC on this model.

Table 14 summarizes the overall hypotheses verification outcomes. Four of the six hypotheses were verified. Two were not verified.

Hypothesis	Outcome
H1: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with at least one of the common types of vulnerabilities with accuracy greater than 80% and precision greater than 80%.	Verified
H2: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as insecure arithmetic flaws with accuracy greater than 80% and precision greater than 80%.	Verified
H3: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as block values as a proxy for time flaws with accuracy greater than 80% and precision greater than 80%.	Not Verified
H4: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as reentrancy flaws with accuracy greater than 80% and precision greater than 80%.	Not Verified
H5: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as locked Ether flaws with accuracy greater than 80% and precision greater than 80%.	Verified
H6: Cloud-based automated machine learning can be used to build a model that identifies smart contracts with vulnerabilities classified as transaction order dependency flaws with accuracy greater than 80% and precision greater than 80%.	Verified

Table 14—Verification of Hypotheses

5.1.2 Response to Research Questions

Research Question 1 asks whether cloud-based AutoML can be used to build effective smart contract vulnerability detection models. According to the results, models can be built using labeled smart contract source code for general vulnerability detection and targeted detection of insecure arithmetic, locked Ether, and transaction order dependency flaws. However, AutoML did not generate acceptable models for those trained to detect reentrancy and block values as a proxy for time flaws. From the results, it can be surmised that there is a correlation between training sample size and model effectiveness. Models trained using vulnerability datasets containing nearly a thousand or more vulnerable samples, such as with general vulnerability detection, insecure arithmetic, locked Ether, and transaction order dependency flaws, achieved acceptable rates of accuracy and precision. However, models trained using datasets with limited examples of vulnerable smart contracts, as with reentrancy and block values as a proxy for time flaws, were not able to achieve acceptable rates of precision.

Research Question 2 asks whether smart contract source code in its native form without an intermediate representation can be used to build effective vulnerability detection models. The results demonstrated that effective vulnerability detection models can be built solely using native smart contract source code for general vulnerability detection, insecure arithmetic, locked Ether, and transaction order dependency flaws.

Research Question 3 asks how quickly cloud-based AutoML-built smart contract vulnerability detection models can identify vulnerable smart contracts. The results demonstrated that the models could complete an evaluation of a smart contract for vulnerabilities in approximately 0.03 seconds. This is exponentially faster than the 2.1

seconds to 352.2 seconds average durations observed by Qian et al. (2022) with traditional static analysis detection tools.

Research Question 4 asks whether there are any advantages to using specific types of compute capabilities when using cloud-based AutoML to build and test smart contract vulnerability detection models. Both GPUs and CPUs were explored in this praxis. As depicted in the Chapter 4 results, GPU-based model training was faster than CPU-based training in all experiments except for the reentrancy flaws experiment. GPU-based testing was also generally faster (the only exception was the VulnerableOrNot test). However, there was no meaningful difference between the two in testing: both CPUs and GPUs took on average about 0.03 seconds to test each smart contract.

5.2 Conclusion

This praxis demonstrated that cloud-based AutoML can be used to build vulnerability detection models based on native smart contract source code for certain types of vulnerabilities. Accuracy and precision of over 80% were achieved for detection models focused on general vulnerability detection, insecure arithmetic, locked Ether, and transaction order dependency flaws. However, the experiments were not successful in demonstrating effective models focused on detecting smart contracts with reentrancy and block values as a proxy for time flaws. The research demonstrated time savings when equipping AutoML with GPU-based compute clusters to build models. However, this was not consistently observed across all AutoML experiments. During model evaluation, there were only negligible differences between the two compute cluster types when testing smart contracts for vulnerabilities. Both CPU- and GPU-based machine learning-generated models were able to test each smart contract for vulnerabilities quickly, with an

average speed of about 3 hundredths of a second. This is exponentially faster than traditional static and dynamic analysis tools and comparable to detection models built using traditional machine learning.

5.3 Contributions to Body of Knowledge

The literature review illustrated gaps in existing research into the use of AutoML to build vulnerability detection models based on smart contract source data. This praxis narrows the gaps by making the following contributions to the body of knowledge:

1. Adds literature on the use of cloud-based AutoML to build and evaluate smart contract vulnerability detection models.
2. Establishes the use of Microsoft Azure AutoML to build and evaluate models that can identify vulnerable smart contracts.
3. Demonstrates the use of raw and full-text smart contract source code without the added processing step of extracting an intermediate representation to build smart contract vulnerability detection models.
4. Exhibits the use of the ScrawlD dataset to build and evaluate cloud-based AutoML vulnerability detection models.
5. Provides statistical data on the performance of cloud-based AutoML-built models when detecting common smart contract vulnerabilities.

5.4 Recommendations for Future Research

It is recommended that future research examine the following areas:

- This praxis focused on the use of AutoML to detect five types of smart contract vulnerabilities. However, there are numerous other known smart contract vulnerabilities. Additional research should be conducted to study the effectiveness

of using AutoML to build vulnerability detection models to identify smart contracts with other vulnerabilities. There are also opportunities to explore the use of AutoML to build vulnerability detection models for other types of software.

- This work evaluated binary classification. Multilabel classification should be studied to determine whether a single detection model can effectively discern and identify distinct types of vulnerabilities that may be present in a smart contract. In addition, researchers should study the use of emerging artificial intelligence technologies, such as generative artificial intelligence, to build detection tools that can identify smart contract vulnerabilities.
- This praxis used the ScrawlD dataset to conduct testing. Other datasets could be explored to determine any differences in effectiveness when used to build vulnerability detection models. A compilation of information from several datasets could perhaps produce more effective detection models.
- The hypothesis statements developed for this study focused on accuracy and precision as key metrics for determining model effectiveness. However, it is clear that other metrics such as recall, F1 Score and MCC are helpful towards providing a more comprehensive evaluation of a model's effectiveness. Further study should be conducted to determine the appropriate set of metrics and thresholds for evaluating the effectiveness of AutoML generated vulnerability detection models.
- The use of other AutoML platforms to build models could be evaluated. In this praxis, Microsoft Azure was used. However, numerous other AutoML platforms are available. Studies should be conducted to determine whether there are differences between the various offerings.

References

Akca, S., Rajan, A., & Peng, C. (2019). SolAnalyser: A framework for analysing and testing smart contracts. In 2019 26th Asia-Pacific Software Engineering Conference (APSEC) (pp. 482-489). IEEE.

Alchemy. (2022). Ethereum Statistics (2022). Retrieved July 11, 2023 from https://www.alchemy.com/overviews/ethereum-statistics

Alchemy. (2022). Learn Solidity: What is a reentrancy attack?, Retrieved February 26, 2023 from https://www.alchemy.com/overviews/reentrancy-attack-solidity

Alharby, M., & Van Moorsel, A. (2017). Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372.

Alkhalifah, A., Ng, A., Chowdhury, M. J. M., Kayes, A. S. M., & Watters, P. A. (2019). An empirical analysis of blockchain cybersecurity incidents. In 2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE) (pp. 1-8). IEEE.

Antonopoulos, A. M., & Wood, G. (2018). Mastering Ethereum: Building Smart Contracts and Dapps O'reilly Media.

Ashizawa, N., Yanai, N., Cruz, J. P., & Okamura, S. (2021). Eth2Vec: learning contract-wide code representations for vulnerability detection on Ethereum smart contracts In Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure (pp. 47-59).

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts (sok). In Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and

Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6 (pp. 164-186). Springer Berlin Heidelberg

Behan, S. (2022). Solidity Smart Contract Testing with Static Analysis Tools (master's thesis, Dublin, National College of Ireland).

Beillahi, S. M., Keilty, E., Nelaturu, K., Veneris, A., & Long, F. (2022). Automated Auditing of Price Gouging TOD Vulnerabilities in Smart Contracts in the 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC) (pp. 1-6). IEEE. Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper, 3(37), 2-1.

Chen, H., Pendleton, M., Njilla, L., & Xu, S. (2020). A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Computing Surveys (CSUR), 53(3), 1-43.

Chibuzor, M. (2022). Smart contract development: Common mistakes to avoid. LogRocket. Retrieved from February 25, 2023 from https://blog.logrocket.com/smart-contract-development-common-mistakes-avoid

Chicco, D., & Jurman, G. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation BMC genomics, 21, 1-13.

ConsenSys. (2018). Ethereum smart contract best practices, Recommendations for smart contract security in solidity. Retrieved February 26, 2023 from https://consensys.github.io/smart-contract-best-practices

Cruz, Y. J., Villalonga, A., Castaño, F., Rivas, M., & Haber, R. E. (2023). Automated Machine Learning Methodology for Optimizing Production Processes in Small and Medium-Sized Enterprises. Available at SSRN 4334026.

Cysneiros Aragão, M. V., Guimarães Afonso, A., Ferraz, R. C., Gonçalves Ferreira, R., & Gomes Leite, S. (2023). A practical evaluation of AutoML tools for binary, multiclass, and multilabel classification. TechRxiv.

Dika, A., & Nowostawski, M. (2018). Security vulnerabilities in Ethereum smart contracts. In the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical, and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 955-962). IEEE.

Durieux, T., Ferreira, J. F., Abreu, R., & Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 530-541).

Eshghie, M., Artho, C., & Gurov, D. (2021). Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In Evaluation and Assessment in Software Engineering (pp. 305-312).

Etherscan. (2023). Etherscan API documentation. Retrieved March 3, 2023 from https://docs.etherscan.io

Exploratory. (2023). Exploratory Desktop. Retrieved March 4, 2023 from https://exploratory.io/

Ferretti, S., & D'Angelo, G. (2020). On the ethereum blockchain structure: A complex network theory perspective Concurrency and Computation: Practice and Experience, 32(12), e5493.

Fontein, R. (2018). Comparison of static analysis tooling for smart contracts on the EVM. In 28th Twente Student Conference on IT.

Gao, Z. (2020). When deep learning meets smart contracts. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (pp. 1400-1402).

Gao, Z., Jayasundara, V., Jiang, L., Xia, X., Lo, D., & Grundy, J. (2019). Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 394-397). IEEE.

Ghaleb, A. (2022). Towards Effective Static Analysis Approaches for Security Vulnerabilities in Smart Contracts. In 37th IEEE/ACM International Conference on Automated Software Engineering (pp. 1-5).

Ghaleb, A., & Pattabiraman, K. (2020). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 415-427).

Guan, C., Zhang, Z., Li, H., Chang, H., Zhang, Z., Qin, Y., ... & Zhu, W. (2021). Autogl: A library for automated graph learning. arXiv preprint arXiv:2104.04987.

Hacken. (2022). Top pitfalls of Web 3.0 Smart Contracts Security. Retrieved February 26, 2023 from https://hacken.io/insights/top-pitfalls-of-web-3-0-smart-contracts-security

Hanussek, M., Blohm, M., & Kintz, M. (2020). Can AutoML outperform humans? An evaluation on popular OpenML datasets using AutoML Benchmark. In 2020 2nd international conference on artificial intelligence, robotics, and control (pp. 29-32).

He, D., Wu, R., Li, X., Chan, S., & Guizani, M. (2023). Detection of Vulnerabilities of Blockchain Smart Contracts. IEEE Internet of Things Journal.

Heimbach, L., & Wattenhofer, R. (2022). SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. arXiv preprint arXiv:2203.11520.

Hossin, M., & Sulaiman, M. N. (2015). A review on evaluation metrics for data classification evaluations. International Journal of Data Mining & Knowledge Management Process, 5(2), 1.

Huang, Y., Bian, Y., Li, R., Zhao, J. L., & Shi, P. (2019). Smart contract security: A software lifecycle perspective. IEEE Access, 7, 150184-150202.

IBM. (2023). What is machine learning?. Retrieved March 7, 2023 from https://www.ibm.com/topics/machine-learning

Jeon, S., Lee, G., Kim, H., & Woo, S. S. (2021). Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using BERT. In KDD Workshop on Programming Language Processing.

Kavitha, M., & Prabhavathy, P. (2021). A review on machine learning techniques for text classification. In 2021 4th International Conference on Computing and Communications Technologies (ICCCT) (pp. 605-610). IEEE.

Liao, J. W., Tsai, T. T., He, C. K., & Tien, C. W. (2019). Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In 2019 Sixth International Conference on Internet of Things: Systems, Management, and Security (IOTSMS) (pp. 458-465). IEEE.

Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 254-269).

Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. NIST.

Mi, F., Wang, Z., Zhao, C., Guo, J., Ahmed, F., & Khan, L. (2021). VSCL: Automating Vulnerability Detection in Smart Contracts with Deep Learning. In 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC) (pp. 1-9). IEEE.

Microsoft. (2023). Azure Machine Learning documentation. Retrieved February 28, 2023 from https://learn.microsoft.com/en-us/azure/machine-learning/

Miller, A., Cai, Z., & Jha, S. (2018). Smart contracts and opportunities for formal methods. In International Symposium on Leveraging Applications of Formal Methods (pp. 280-299). Springer, Cham.

Mitchell, T. M. (1997). Machine learning. New York: McGraw-Hill.

Momeni, P., Wang, Y., & Samavi, R. (2019). Machine learning model for smart contracts security analysis. In 2019 17th International Conference on Privacy, Security and Trust (PST) (pp. 1-6). IEEE.

Mueller, B. (2018). Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam, 9, 54.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review, 21260.

National Institute of Standards and Technology. (2023). Computer Security Resource Center Glossary. Retrieved June 14, 2023 from https://csrc.nist.gov/glossary

Nick, S. (1996). Smart contracts: Building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought, 18(16), 2.

Noshiri, N., Khorramfar, M., & Halabi, T. (2021). Machine Learning-as-a-Service Performance Evaluation on Multi-class Datasets. In 2021 IEEE International Conference on Smart Internet of Things (SmartIoT) (pp. 332-336). IEEE.

Pachmann, T. (2022). An evaluation and comparison of AutoML solutions: Azure AutoML and EvalML. Hochschule Darmstadt.

Padurariu, C., & Breaban, M. E. (2019). Dealing with data imbalance in text classification. Procedia Computer Science, 159, 736-745.

Palladino, S. (2017). The Parity Wallet Hack Explained. OpenZepplin. Retrieved January 16, 2023, from https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

Pandey, H. (2023). Preventing Arithmetic Overflow and Underflow in Solidity: Best Practices and Solutions. Medium. Retrieved February 26, 2023, from

https://medium.com/coinmonks/preventing-arithmetic-overflow-and-underflow-in-solidity-best-practices-and-solutions-bc06ea6da9fd

Patibandla, R. L., Srinivas, V. S., Mohanty, S. N., & Pattanaik, C. R. (2021). Automatic machine learning: An exploratory review. In 2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO) (pp. 1-9). IEEE.

Patil, P. (2018). What is Exploratory Data Analysis?. Towards Data Science. Retrieved March 4, 2023 from https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15

Perez, D., & Livshits, B. (2019). Smart contract vulnerabilities: Does anyone care?. arXiv preprint arXiv:1902.06710, 1-15.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., & Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. IEEE Access, 7, 78194-78213.

Qian, P., Liu, Z., He, Q., Huang, B., Tian, D., & Wang, X. (2022). Smart contract vulnerability detection technique: A survey. arXiv preprint arXiv:2209.05872.

Qian, S., Ning, H., He, Y., & Chen, M. (2022). Multi-Label Vulnerability Detection of Smart Contracts Based on Bi-LSTM and Attention Mechanism. Electronics, 11(19), 3260.

REKT Database. (2023). Retrieved February 26, 2023, from https://de.fi/rekt-database

Rossini, M., Zichichi, M., & Ferretti, S. (2022). Smart Contracts Vulnerability Classification through Deep Learning. In Proceedings of the Twentieth ACM Conference on Embedded Networked Sensor Systems (pp. 1229-1230).

Sharma, N. (2022). A Survey of Mythril, A Smart Contract Security Analysis Tool for EVM Bytecode.

Smart Contract Weakness Classification Registry. (2023). Retrieved January 16, 2023, from https://swcregistry.io

Sürücü, O., Yeprem, U., Wilkinson, C., Hilal, W., Gadsden, S. A., Yawney, J., ... & Giuliano, A. (2022). A survey on ethereum smart contract vulnerability detection using machine learning. Disruptive Technologies in Information Sciences VI, 12117, 110-121.

Tharwat, A. (2020). Classification assessment methods. Applied Computing and Informatics, 17(1), 168-192.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). Smartcheck: Static analysis of Ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (pp. 9-16).

Truong, A., Walters, A., Goodsitt, J., Hines, K., Bruss, C. B., & Farivar, R. (2019). Towards automated machine learning: Evaluation and comparison of AutoML approaches and tools. In 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI) (pp. 1471-1479). IEEE.

Usman, T. A., Selçuk, A. A., & Özarslan, S. (2021). An Analysis of Ethereum Smart Contract Vulnerabilities. In 2021 International Conference on Information Security and Cryptology (ISCTURKEY) (pp. 99-104). IEEE.

Vivar, A. L., Castedo, A. T., Orozco, A. L. S., & Villalba, L. J. G. (2020). An analysis of smart contracts security threats alongside existing solutions. Entropy, 22(2).

Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., & Yang, X. (2021). Smart contract security: a practitioners' perspective. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1410-1422). IEEE.

Wang, W., Song, J., Xu, G., Li, Y., Wang, H., & Su, C. (2020). Contractward: Automated vulnerability detection models for Ethereum smart contracts. IEEE Transactions on Network Science and Engineering, 8(2), 1133-1144.

Wang, X., Sun, J., Hu, C., Yu, P., Zhang, B., & Hou, D. (2022). EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection. Wireless Communications and Mobile Computing, 2022.

Waring, J., Lindvall, C., & Umeton, R. (2020). Automated machine learning: Review of the state-of-the-art and opportunities for healthcare. Artificial intelligence in medicine, 104, 101822.

Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., & Grunske, L. (2022). Vudenc: Vulnerability detection with deep learning on a natural codebase for python. Information and Software Technology, 144, 106809.

Wohrer, M., & Zdun, U. (2018). Smart contracts: security patterns in the Ethereum ecosystem and Solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 2-8). IEEE.

Wu, H., Dong, H., He, Y., & Duan, Q. (2023). Smart Contract Vulnerability Detection Based on Hybrid Attention Mechanism Model. Applied Sciences, 13(2), 770.

Xu, Y., Hu, G., You, L., & Cao, C. (2021). A novel machine learning-based analysis model for smart contract vulnerability. Security and Communication Networks, 2021, 1-12.

Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y. F., Tu, W. W., ... & Yu, Y. (2018). Taking human out of learning applications: A survey on automated machine learning. arXiv preprint arXiv:1810.13306.

Yashavant, C. S., Kumar, S., & Karkare, A. (2022). ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities. arXiv preprint arXiv:2202.11409.

Zavarella, L. (2020). A Review of Azure Automated Machine Learning (AutoML). Medium. Retrieved March 4, 2023, from https://medium.com/microsoftazure/a-review-of-azure-automated-machine-learning-automl-5d2f98512406

Zhang, W., Banescu, S., Pasos, L., Stewart, S., & Ganesh, V. (2019). MPro: combining static and symbolic analysis for scalable testing of smart contract. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE) (pp. 456-462). IEEE.

Zeng, Q., He, J., Zhao, G., Li, S., Yang, J., Tang, H., & Luo, H. (2022). EtherGIS: A Vulnerability Detection Framework for Ethereum Smart Contracts Based on Graph Learning Features. In 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 1742-1749). IEEE.

Zheng, Z., Xie, S., Dai, H. N., Chen, X., & Wang, H. (2018). Blockchain challenges and opportunities: A survey. International Journal of Web and Grid Services, 14(4), 352-375.

Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., & Imran, M. (2020). An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems, 105, 475-491.

Appendix A

ScrawlD Dataset Vulnerability Label	Vulnerability Type	Prepared Datasets					
		Insecure Arithmetic Flaws Dataset	Block Values as a Proxy for Time Flaws Dataset	Reentrancy Flaws Dataset	Locked Ether Flaws Dataset	Transaction Order Dependency Flaws Dataset	VulnerableOrNot Dataset
ARTHM	Insecure Arithmetic	•					•
LE	Locked Ether				•		•
RENT	Reentrancy			•			•
TimeM	Block Values as a Proxy for Time		•				•
TimeO	Transaction Order Dependency					•	•
UE	Unchecked Call Return Value	This vulnerability type was not studied in this praxis.					
TX-Origin	Authorization through tx.origin	This vulnerability type was not studied in this praxis.					

Table 15—Mapping of ScrawlD Dataset Vulnerability Labels to Prepared Datasets