CryptoProbe: Identification & Curtailment of Honeypots on Decentralized Blockchain Networks using Machine Learning and Code Analysis

BTech Project Report,

submitted in partial fulfillment of the requirement for the award of

Integrated Post Graduate (IPG) Master of Technology degree in Information Technology

by

Ojaswa Sharma (2017IMT-102)

under the supervision of

Prof. Shashikala Tapaswi
Dr. K K Pattanaik

विश्वजीवनामृतं ज्ञानम्

ABV - INDIAN INSTITUTE OF INFORMATION TECHNOLOGY AND MANAGEMENT GWALIOR - 474015

2020

CANDIDATES DECLARATION

I hereby certify that the work, which is being presented in the report, entitled **Identification & Curtailment of Honeypots on Decentralized Blockchain Networks using Machine Learning and Code Analysis**, in partial fulfillment of the requirement for the award of the Degree of **Integrated Post Graduate (IPG) Master of Technology** and submitted to the institution is an authentic record of my own work carried out during the period *July 2020* to *October 2020* under the supervision of **Prof. Shashikala Tapaswi** and **Dr. K K Pattanaik**. We also cited the reference about the text(s)/figure(s)/table(s) from where they have been taken.

Date: Signatures of the Candidates

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Date: Signatures of the Research Supervisors

ABSTRACT

Modern day blockchains enable the concept of "Smart Contracts”, which are programs executed across a decentralized network of nodes which allow commitments comprising in the transfer of tokens or cryptocurrencies when certain conditions are met. The rise in popularity of cryptocurrencies and smart contracts has made them an interesting target for attackers. While the traditional approach violates vulnerabilities in Smart Contracts, a new pattern towards a more proactive methodology is on the rise, where attackers do not look for for exploitable contracts anymore but instead, they themselves try to lure their users (victims) into traps by deploying seemingly vulnerable contracts that, upon execution, unfurl hidden traps. These new types of contracts are commonly referred to as honeypots. We develop CryptoProbe - A SaaS Platform that employs symbolic execution, well-defined heuristics and data science techniques to enable large-scale, robust and efficient analysis of Smart Contracts on Ethereum Blockchain Network to identify honeypots. The SaaS wrapper ensures integration of CryptoProbe with existing wallets, APIs, and tools as a built-in feature or plugin.

Keywords: cryptocurrency, smart contracts, honeypots, security, symbolic execution, SaaS, data science, machine learning, classification, heuristics, ethereum, blockchain

ACKNOWLEDGEMENTS

I am highly indebted to **Prof. Shashikala Tapaswi** and **Dr. K K Pattanaik**, and are obliged for giving me the autonomy of functioning and experimenting with ideas. I would like to take this opportunity to express my profound gratitude to them not only for their academic guidance but also for their personal interest in my project and constant support coupled with confidence boosting and motivating sessions which proved very fruitful and were instrumental in infusing self-assurance and trust within me. The nurturing and blossoming of the present work is mainly due to their valuable guidance, suggestions, astute judgment, constructive criticism and an eye for perfection. My mentors always answered myriad of my doubts with smiling graciousness and prodigious patience, never letting me feel that I am a novice by always lending an ear to my views, appreciating and improving them and by giving a free hand in my project. It's only because of their overwhelming interest and helpful attitude, the present work has attained the stage it has.

Finally, I am grateful to our Institution and colleagues whose constant encouragement served to renew my spirit, refocus my attention and energy and helped me in carrying out this work.

(Ojaswa Sharma)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DAO	Decentralized Autonomous Organization
EVM	Ethereum Virtual Machine
SMT	Satisfiability Modulo Theories
SaaS	Service as a Service
ASP	Application Service Provider
TOD	Transaction-Ordering Dependence
ID	Inheritance Disorder
SMC	Straw Man Contract
SESL	Skip Empty String Literal
BD	Balance Disorder
US	Uninitialized Struct
TDO	Type Deduction Overflow
HSU	Hidden State Update
API	Application Programming Interface
CFG	Control Flow Graph
SHA	Secure Hash Algorithms
SSH	Secure Shell
DFS	Depth First Search
AUROC	Area Under the Receiver Operating Characteristics
ROC	Receiver Operating Characteristics
TPR	True Positive Rate
FPR	False Positive Rate

NOTATIONS

σ	World state
σ'	Modified world state
σ[a]	State of account *a*
σ[a]b	Balance of account or contract *a*
σ[a]s	Storage of contract *a*
I	Transaction execution environment
Ti, Tj	Transactions
Ia	Address of smart contract being executed
Id	Transaction input data
Iv	Transaction value
Is	Transaction sender

CHAPTER 1

Introduction

The idea of blockchain was presented in 2009 with the arrival of Nakamoto Satoshi's Bitcoin [2], which developed enormously from that point forward, to an extent that it is regarded as one of the most ingenious advances since the innovation of the Internet itself. Modern blockchains like Ethereum [3] aim to decentralize transactions and programs as a whole through so-called smart contracts which are deployed, executed or invoked and removed from the blockchain via transactions. Ethereum has confronted several annihilating attacks [4, 5] caused on vulnerable smart contracts, for example, The DAO Hack (2016) and The Parity Wallet Hack (2017). Malicious contracts that that look defenseless however are exploitative are a rising pattern. Recently, cyber attackers and criminals have begun utilizing these contracts to beguile users by deploying contracts that pretend to give away funds or services, but in actually contain hidden traps. These types of attacks were then named by the community as "Honeypots" [6, 7, 8].

1.1 Background

1.1.0.1 Smart Contracts

The thought of smart contracts has been presented by Nick Szabo in 1997 [9]. He depicted the idea of a trust-less framework comprising of self-executing computer programs that would encourage the advanced confirmation (digitally) and requirement of all agreement provisos contained in lawful contracts. This was only a concept until 2015 when Ethereum was released and only then stood the first chance of this idea converting into a reality. Ethereum smart contracts are not the same as usual programs in several aspects. For instance, like every transaction, the code stored on the blockchain is immutable its execution is ensured by the blockchain. While they are immutable, they can also be annihilated, only in the event that they contain the vital code programmed to handle their demolition, after which, the agreement cannot be executed and its funds are moved to another address. Like other languages, these contracts are also developed using high-level programming languages which are compiled into low-level byte-code to be deployed and executed via transactions. Each smart contract is then identified by a 160-bit address. Despite a huge assortment of programming dialects (such as LLL[10], Vyper [11], and Bamboo[12]), Solidity[13] remains the most unmistakable programming language for creating smart contracts in Ethereum. Solidity's semantics and syntax is like a combination of C and JavaScript. It accompanies a huge number of exceptional ideas that are explicit to smart contracts, for example, the exchange of funds or the capacity to call different contracts.

1.1.0.2 Ethereum Virtual Machine

The Ethereum blockchain comprises a network of mutually distrustful hubs (or nodes) which together structure a decentralized public record, called 'Ledger'. This ledger permits clients to make and summon smart contracts by submitting exchanges to the framework. 'Miners', who process transactions, execute contracts while block verification using a devoted virtual machine referred to as 'Ethereum Virtual Machine' [3]. The EVM is a stack-based, register-less virtual machine, running low-level bytecode, represented by an instruction set of opcodes. An idea of 'gas' is acquainted with ensure an agreement's end and keep miners from being stuck in never-ending execution loops. It partners expenses to the execution of every instruction. When a transaction is initiated, the sender is required to indicate the total gas that the sender is happy to pay for the execution of the concerned contract. The execution of a smart agreement brings about an adjustment of the world state σ, an information structure put away on the blockchain planning an address to a record state σ[a]. The record condition of a smart agreement comprises of two principal parts: an equalization σ[a]b, which holds the measure of ether possessed by the agreement, and capacity σ[a]s, which has the constant information of the agreement. From a higher-level, the Ethereum Virtual Machine can basically be viewed as a transaction-based state machine that takes in I and σ as input, and yields a changed (new/modified) world state σ'.

1.1.0.3 Etherscan Blockchain Explorer

Etherscan [14] is an online-available platform that gathers and shows blockchain explicit data, as well as provide APIs to interact and request the available data. It goes about as a blockchain pilot permitting clients to handily look into the substance of individual blocks, transactions, and smart contracts on Ethereum. Furthermore, It offers numerous functions apart from investigating capacities. One of these is the opportunities for owners to distribute their source code and affirm that the byte-code put away under a particular address is the aftereffect of the accumulation of the predefined source code. This establishes an assertion to transparency between compiled code and the higher level language. The users are also allowed to leave suggestion, comments and remarks on added smart contracts.

1.1.1 Motivation

Blockchain is public and hence all users of a blockchain can view its smart contracts. Certainly enough, this prompts a circumstance where bugs, including security openings, are noticeable to each user and they may not be fixed immediately. When we specifically consider Ethereum-based smart contracts, they incorporate ambiguities and simple yet uncertain builds in its higher-level language, Solidity, EVM bugs, compiler discrepancies, bug immutability and lack of focal source to document known issues, bugs, vulnerabilities, attacks[15] and discrepant constructs. These malicious smart contracts (honeypots) have a common characteristic, they quite often attempt to appear as though they were planned by a novice in an attempt for pursuers to trap the seemingly new contestant. Considering this, it also spots the entanglements that can come upon a new entrant to the space, and fill in as a fascinating contextual analysis into the wild-west universe of smart agreement security. By dissecting a couple of the additionally intriguing instances of not really weak contracts, we can understand a profound comprehension of how smart contract security works.

This works as a motivation to create and deploy an efficient and robust system to help users detect malicious Honeypots and save them from fraudulent transactions. With a rising trend in decentralized networks and a global shift to the usage of Smart Contracts, a curb against these types of attacks is utmost necessary. Unfortunately, the significant lack of research, identification techniques and deployable products to counter Smart Contract vulnerabilities [16, 17, 18, 19] have continued to encourage attackers to follow these kinds of malpractices.

1.1.2 Honeypots

Honeypot Definition[20] *"A honeypot is a smart contract that pretends to leak its funds to an arbitrary user (victim), provided that the user sends additional funds to it. However, the funds provided by the user will be trapped and at most the honeypot creator (attacker) will be able to retrieve them."*

As the context has already been established with previous text, it must be evident how honeypots come to effect in daily life. Honeypots lure users (victims) by setting up traps which makes the users believe that they will gain additional funds after they transact with the smart contract. While the devious nature is not evident, the user as per his own understandings about source code and smart contracts, deduce to this decision. However, it is not obvious to users that will uncover secret conditions to make all or at the least, most of the transferred funds to stay in the honeypot itself.

1.1.3 Symbolic Execution

It is an analysis technique to assert if certain conditions and properties are satisfied by programs or software [21, 22, 23, 24, 25]. Symbolic execution was particularly brought-in in the mid 1970s to check if programs violated certain conditions or properties. Some examples of these conditions can be, restriction of division by 0, Aspects of interest could be that no division by zero is ever performed, access of value behind NULL pointer, no authentication bypasses except the legitimately defined process, etc. There exists a gap of automated workflows to assert these properties, we can leverage existing analyses and heuristics to determine them in certain environments in real-life, especially with respect to privacy, security and mission-critical applications.

The execution is handled by a *symbolic* execution engine, whose job is to accumulate and manage each control flow path, by maintaining: (i) a Boolean equation (first-order) portrays fulfilled branch conditions along the corresponding path, and (ii) *symbolic* memory store that maintains a map of various factors or variables to symbolic equations, expressions, qualities or values. The execution of each branch is to refresh the formula, as the tasks or assignments update the symbolic map store. Satisfiability Modulo Theories (SMT) Solvers [26, 27] are used to verify satisfiability of conditions along each explored path of the Control Flow Graphs, i.e if each path is realizable. This component which uses SMT solvers to to deduce feasibility of paths (if the path's equation is fulfilled by assigning concrete values to the piece of code's/equation's symbolic contentions), is referred as a model checker.

1.1.4 Data Science and Machine Learning

Data Science or Data Analytics is a control that draws procedures from various fields like arithmetic, measurements, software engineering, and data science, with the objective of finding bits of knowledge from true data. The term is firmly identified with different fields, for example, machine learning, artificial intelligence, business intelligence and data mining. Machine Learning is the sub-field of Artificial Intelligence that gives PC programs, software and tools the capacity to improve from feedback without the need of being customized or modified [28]. Machine Learning algorithms can be classified as supervised and unsupervised. The prior can recognize designs and patters that links *features* (quantifiable qualities) to *labels* (a specific property). The *training* or learning refers to the process in which the learning algorithms identify patterns from known examples (data classified by labels). The result of the learning process is a model, which is an estimation of the latent links between features to labels. The *testing* step, unknown names are appointed, which were no the part of learning stage in order to deduce the effectiveness of the predictions made by the trained model with an intention of determining results with concealed cases. If each label refers to the subset of possible classes, supervised machine learning is referred to as *classification*. For instance, a model trained to classify a smart contract as honeypot or non-honeypot is an example of classification problem, which in-turn being a part of supervised machine learning.

1.1.5 Zero-day Vulnerability

As the name suggests, "zero-day" name is derived by the fact that the vulnerability or issue or bug in concern has just been discovered and the development, design or security teams have spent 0 days to fix the exposed issue. Hence, the expression alludes to a newfound programming vulnerability which is yet unfixed and might be already being exploited by malicious users and attackers. It is important to keep these issues private and in case, it is publicly acknowledged, the code owners (sellers) need to effectively and swiftly deploy a patch to the problem to protect its services, resources and users. In any case, if the product owners fail[29] to deliver a fix before malicious users and attackers begin to leverage the security opening, it is then referred as a **Zero-day Attack**.

1.1.6 Service as a Service

Software as a Service (SaaS) [30] is a distribution architecture which involves a provider to host applications or services and provide them to clients over the Web (or Internet). Distributed computing establishes 3 main paradigms of service exposure, Infrastructure as a Service, Service as a Service and Platform as a Service (IaaS, SaaS and PaaS). SaaS is usually directly related to on-demand delivery models which can facilitate a services like computing. The SaaS application management model is similar to Application Service Providers (ASP), where the supplier can fulfill the customer's software requirements and provides the application to approved end users over Internet. In the on-demand SaaS model, the supplier gives end-users network-based access to an isolated duplicate (in case of software) and/or access to the processes (in case of service) of an application created particularly for SaaS appropriation. The source-code is usually same for all users (except for per-tier features and constraints), and new fixes and features are delivered to the clientele together.

1.2 Literature Survey

1.2.1 Security Bugs in Contracts

Authors in [1] discovered several security problems which can alter the execution of Ethereum based smart contracts which can arise discrepancies between expected and actual transactions involved in the contracts. Their study in open-ended and distributed networks showed unpretentious gaps between the execution and understandings of semantics of primary platform and services.

1.2.1.1 Transaction-Ordering Dependence

In each 'block' of a blockchain, there exists multiple transactions. This means that in each epoch time, the world state of the blockchain is updated multiple number of times. For instance, if multiple transactions (say Ti and Tj) invoke a smart contract at world state σ. In these cases, when a single block contains multiple transactions, the state at the time of invocation of individual transaction is unknown to the user. This order of execution and updates can only be decided by miners executing the blocks. Final state in some contracts can be a factor of order of updates and hence, the final state in these cases is dependent on miners. Contracts with such scenario or conditions are called as TOD Contracts or Transaction-Ordering Dependent Contracts.

1.2.1.2 Timestamp Dependence

In scenarios where a smart contracts may use the timestamp associated with the block to trigger or assert conditions which result in execution of operations, a minute change in timestamp can significantly affect the state and behavior of contracts (if timestamp is used for sending money). In Ethereum, the timestamp can roughly vary by 900 seconds to be accepted by fellow miners. Thus, the final state and behavior of a contract is dependent on the timestamp chosen by those entities. These are referred as Timestamp-Dependent Contracts.

1.2.1.3 Mishandled Exceptions

In Ethereum, similar to any other programming language, a contract can invoke another contract in multiple ways, including send instruction or by directly calling the concerned function from other contract (anotherContract.otherFunction()). Exceptions can arise due to various conditions (such as topped-up call stack, insufficient gas), in which case the contract called stops its execution, reverts the changes made to the initial state and return a boolean value false. This occurrence of exception may or may not get reported to the caller contract, depending on how the failed contract was invoked. For example, when the send instruction is used, the caller contract is required to particularly verify the occurrence of an exception by asserting the return value. Unfortunately, about 30% contracts do not assert returned values in such cases. This inconsistency initiated by not handling exceptions, can further lead to discrepancies in execution of smart contracts.

1.2.1.4 Re-entrancy Vulnerability

Re-entrancy vulnerability is an adverse and well-identified issue which was also the cause behind The Dao Hack [4], where over 3.6 Million Ether (current value of about 1.4 Billion US Dollars) was stolen by exploiting this vulnerability. The previous section describes how one contract can invoke another, however, in the meantime, the execution of the caller contacts is halted until the end of call. The implication of this issue is not evident at the time of creation of contract but can lead to a situation where the beneficiary of the call leverages the intermediate state of caller to perform a malicious or unwanted operation.

1.2.2 Taxonomy of Honeypots

Authors of [20] have studied behavior of honeypots and classified them into eight techniques. We know that Honeypots deceive victims into traps by using funds as a form of bait. The authors provide a brief of these trapdoors used by honeypots a summary of these hidden trapdoors.

1.2.2.1 Balance Disorder (BD)

Users may not have the complete knowledge about the precise moment when the balance of contract is modified. Honeypots utilizing this method vows to return the value invoked with along with its own balance when a certain condition is satisfied, such that when the funds sent are larger than balance. Practically, the funds sent to contract are added to its balance before the execution begins. Therefore, the condition will never be satisfied.

1.2.2.2 Inheritance Disorder (ID)

This technique becomes relevant when there exists 2 contracts defined with an inheritance relationship, both having a variable defined with a common name that is critical for triggering the condition which invokes transfer of funds. While the variable may appear same to victims, they are internally distinct. Accordingly, modification in one does not changes the other.

1.2.2.3 Skip Empty String Literal (SESL)

This technique arises due to a bug in a particular version of Solidity (language in which Smart Contracts are written) Compiler. This bug results in skipping the functional parameter in case it is an empty string literal and shifts the rest without acknowledging the existence of empty literal. This is exploited to transfer funds to a different account than what it seems.

1.2.2.4 Type Deduction Overflow (TDO)

Smart Contracts also support type deductions, where the type of a variable, if not specified, is picked by the compiler. Solidity (or EVM) picks this to the most fitting size (smallest type). This behavior is exploited to trigger type overflows, resulting in unexpected behaviors.

1.2.2.5 Uninitialized Struct (US)

Similar to languages such as C++, GoLang, Solidity also allow creation of composite datatypes using Structs. By default, of a struct is not initialized, the value written to it defaults to the start of storage of that contract. Due to this, any existing value at the beginning is overwritten. In case, the initial values are transfer condition critical, the contract misbehaves.

1.2.2.6 Hidden State Update (HSU)

Internal transactions without any value transfer are inherently hidden in EtherScan. This is leveraged into changing the state of the contract, while making the user believe the existence of a different false state.

1.2.2.7 Hidden Transfer (HT)

Online exploratory platforms like EtherScan shows source code of smart contracts deployed on blockchain. While the intent is to establish trust, the limited view of browsers and poor design of websites, the default view is truncated. Malicious users leverage this to insert some code such that it is placed outside the view window and transfer funds maliciously.

1.2.2.8 Straw Man Contract (SMC)

While deploying, platforms verify the integrity of contracts but not their source code or the code which is further invoked. Straw Man, like scarecrows, as the name suggests, are dummy contracts deployed to beguile users, but exist to divert attention from a malicious contract in place. This is often achieved by deploying a source code with multiple contracts and one invoking the other, in which case, only one is actually executed which particularly modifies (execute or revert) transactions maliciously.

1.2.3 Honeypot Feature Extraction

The authors in [31, 32] perform various experiments and generate a list of relevant features obtained from EtherScan API which is used to identify important properties such as transitional behavior that can help identify usage of various malicious techniques. The features can be categorized into 3 categories, depending on their source of origination:

a. Features obtained from Source Code

b. Features obtained from Transactions

c. Features obtained from flow of funds

The features are based on some assumptions (verified) and behaviors observed after considering a huge set of data. The authors study various events (including fund flow), deduced variables and their frequency of occurrences to link these statistics to the behavior of honeypots in order to help us generate a behavioral profile to classify honeypots.

1.3 Thesis Objectives and Deliverables

Objectives and deliverables of the thesis work are mentioned in the section below:

1.3.1 Objectives and Deliverables

1. To implement a system to detect and classify various types of honeypots on Smart Contracts.

2. To implement machine learning models in order to make the system capable of detecting new and unknown types of honeypots, including honeypots exploiting zero-day vulnerabilities.

3. To develop an approach to generalize existing implementations and extend them to be more robust and efficient.

4. To expose this implementation in the form of Service as a Service (SaaS) to allow users/developers to integrate the service into their systems and wallets for real-time protection against honeypots.

5. To allow one or more methods of SaaS consumption in the form of Plugin, Application, or API.

CHAPTER 2

Design Details and Implementation

2.1 Phases of Honeypot

To better understand system methodology, it is required to create a high-level behavior of honeypots[20].

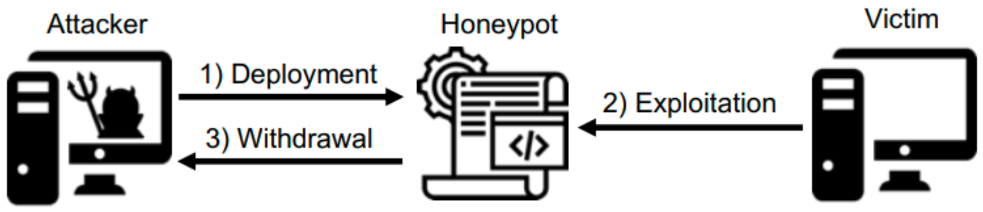


Figure 2.1: Actors and phases of honeypot

Figure 2.1 shows how honeypots operate with entities involved and it can be considered as 3-step process:

1. Malicious user deploys the vulnerable contract with some initial investment as bait on the blockchain;

2. Victim sends funds to the contract in an attempt to exploit the seemingly-vulnerable contract and fails;

3. The attacker (contract owner) withdraws all funds deposited in the contract as result of funds collected from exploited victims.

It is a noticeable insight than an attacker is similar to a regular blockchain user and doesn't need access to any special operations in order to execute these kind of attacks.

2.2 System Design

The project can be viewed as 2 independent components: The **Detection System** and the **SaaS System**, which leverages the first. An overview of the overall system design and architecture is well depicted by Figure 2.2.

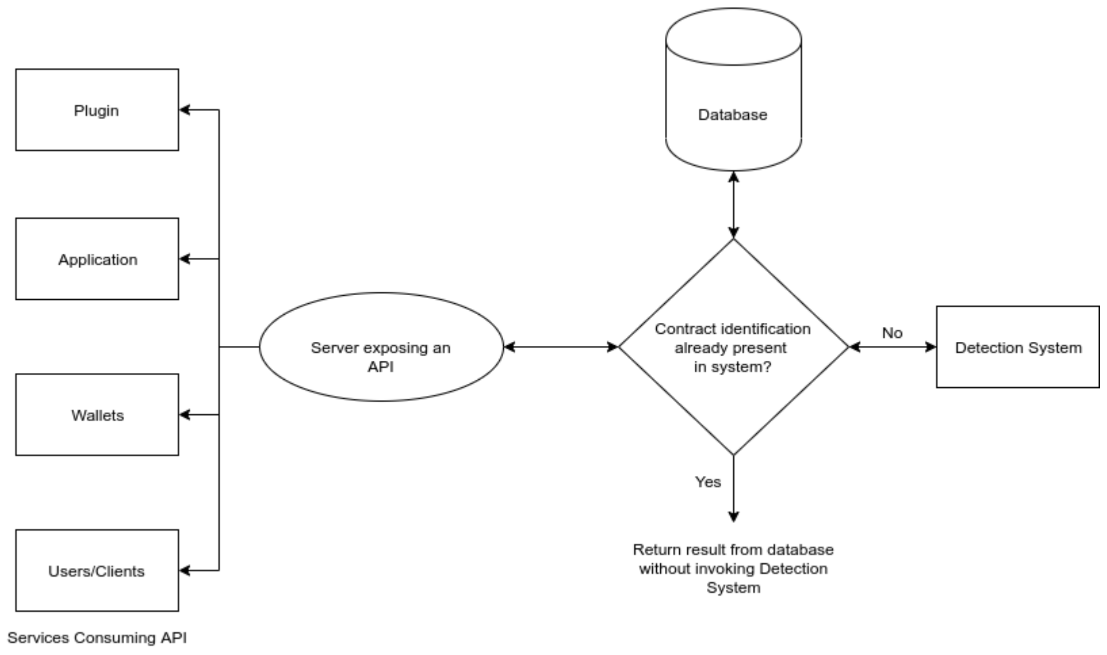

Plugin
Application
Wallets
Users/Clients
Services Consuming API
Server exposing an API
Database
Contract identification already present in system?
No
Detection System
Yes
Return result from database without invoking Detection System

Figure 2.2: Overall architecture of the system

2.3 Detection System

The detection system contains two units, as represented in the Figure 2.3. The Code Analysis system uses static analysis to analyse Ethereum smart contracts and the Machine Learning Model, as the name suggests uses machine learning approach to detect and discover honeypots.

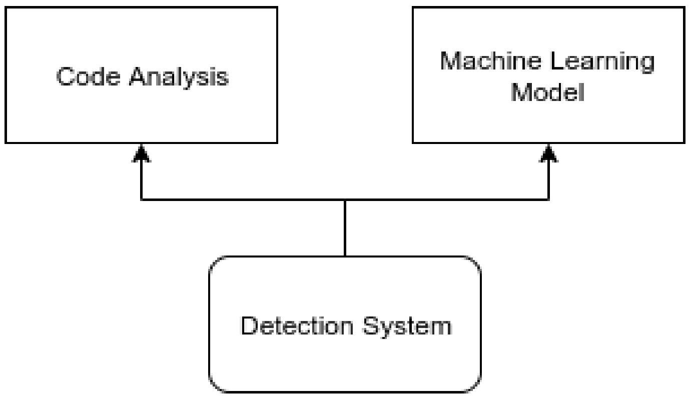


Figure 2.3: High-level view of Detection System

2.3.1 Code Analysis

Figure 2.4 shows an overview of architecture and flow of analysis of the CodeAnalysis component. CodeAnalysis takes EVM byte-code as input and returns a detailed report of multiple techniques of honeypots detected as its output. The component consists of 3 main parts, in order of execution, first **Symbolic Analysis**, followed by **Cash-Flow Analysis**, and **Honeypot Analysis**. Symbolic Analysis creates possible paths and iteratively executes each path in created Control Flow Graph(CFG), the result of which is propagated to the following analysis components. The Cash Flow Analysis leverages the previous result to judge if contract can transfer and receive funds. The Honeypot Analysis Component detects various honeypots according to taxonomy of honeypots along with results of previous sections and feature heuristics. Each component leverages Z3 as *Satisfiability* Modulo Theory [33] Solver to assert constraints.

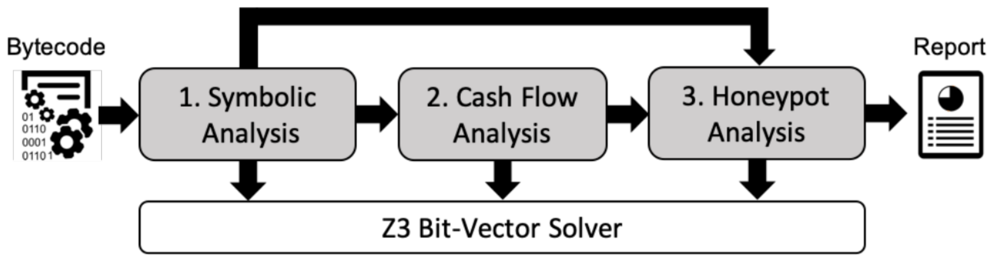


Figure 2.4: An overview of analysis pipeline of the *Code Detection* component

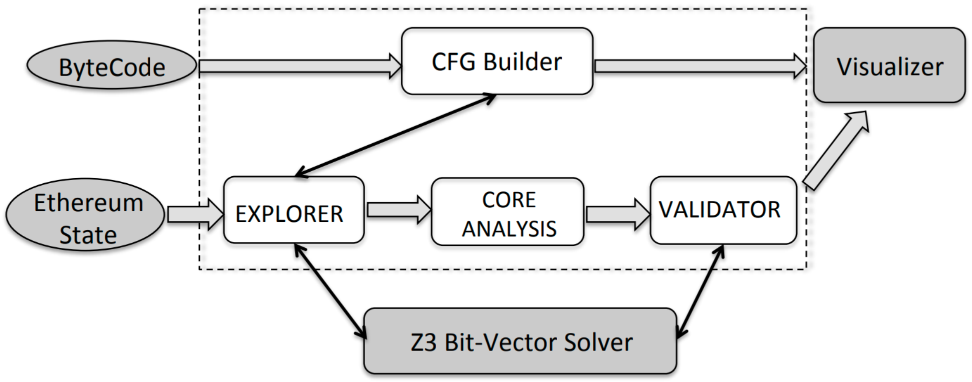


Figure 2.5: Architecture of Symbolic Analysis in The Oyente Tool[1]

As it is evident, the foundation of created analysis tool is Symbolic Execution [22, 1], in which Symbolic expressions for symbolic input values represents actual concrete values of program's variables. Every path is a formula over inputs built by asserting collected constrains, to deduce if the path is satisfiable for execution. The path is said to be feasible only if it is satisfiable, otherwise infeasible. Symbolic Execution can heuristically assert each path of program, which makes it very efficient in deducing predetermined conditions. Hence, for testing any particular case such as TOD, we compare results generating from permutations of multiple execution paths, which is difficult with other approaches (such as dynamic testing) due to complexity and non-determinism in execution behaviors.

The Code Analysis section of the Detection System of the overall service 'Crypto-Probe' is majorly implemented using the programming language Python, with nearly 4,100 lines of code. Following is a brief description of implementation of each sub-component:

2.3.1.1 Symbolic Analysis

As mentioned above, this component uses symbolic execution engine to imitate execution and constructs CFG from bytecode, where each node of the graph refers to a block and thus a edge means a jump linking them. A reused and modified version of the symbolic execution engine proposed by Luu et al. [32, 34] is used. The execution engine comprises of interpreter loop receiving and executing each instruction among a received block. The execution loop halts when a timeout constraint is reached or execution of all blocks is completed. There is a further global time limit which terminates the loop if exceeded. The execution follows DFS approach (Depth-first Search) to explore branches recursively and query Z3 to assert if the path is satisfiable (or feasible).

2.3.1.2 Cash Flow Analysis

A positive honeypot must transfer (receive, and send) funds i.e investment or loot and that explains the motive of this analysis, to improve efficiency of overall detection by eliminating invalid cases that cannot be honeypots. This is distinguished by asserting all possible paths and finding the one p not ending with operation opcode *REVERT*. We can find if the contract can accept funds if the constraint on p is satisfied, i.e. if Iv > 0 is fulfilled. Furthermore, in a same manner, if there exists atleast one path or call ending with operation *SELFDESTRUCT*, which also fulfills constraints, we can determine if a user can transfer funds out of the contract.

2.3.1.3 Honeypot Analysis

This section consist of several sub-sections, each responsible for detection of a technique. These sections are based on the taxonomy of honeypots and each check for satisfiability of symbolic rules established for identification of that type of honeypot. For example, to detect BD: All calls in C are iterated to discover an infeasible basic-block, which satisfies cv = Iv + $\sigma[Ia]b, inwhichcaseitconfirmsthecontractasBalanceDisorderHoneypot.Infutu$

2.3.2 Machine Learning

The Machine Learning model uses existing feature detection (Section 2.3) and exploratory analysis conducted in Camino et al. [31] and create a classification model to identify honeypots.

2.3.2.1 Data Acquisition

Starting from initial blocks, this study is limited to roughly 2 million smart contracts of the Ethereum blockchain. Source Code and Compilation information was occupied for nearly 160,000 contracts using EtherScan API [14] endpoints. Furthermore, a total of about 140M regular and nearly 4.5M internal transactions were also downloaded for these contracts.

2.3.2.2 Labels

The authors presenting taxonomy of honeypots (Section 1.2.2)[20] created a classification tool which labelled honeypots to the kind of technique used, as described in taxonomy. However, to assert efficiency, they also manually verified by inspected the contract source codes for corresponding labels. In case of repetitive or identical contracts, only one of them was used for representation. Contracts were grouped on the basis of SHA-256 calculated from compiled byte-code. Once grouped, labels can be assigned to each honeypot in corresponding group with respect to the representative contract. Only verified labels (after manual inspection and assertion) are used in the process, while ignoring incorrect and false positives.

2.3.2.3 Feature Detection

Like mentioned in the previous sections, features are divided into 3 categories depending on where they originate from. Taking forward the work done by Ramiro Camino et. al [31, 32], the features categorized as Source Code features, Transaction features and Fund Flow features. These features will be used to implement the classification model. The fund flow features are considered the most efficient while identifying existing honeypots i.e. honeypots which are already in action and have exploited victims at-least once; these events which can be used to classify a honeypot, such are:

1. Creation of a valid contract

2. Setting up an initial deposit to the contract as investment (or trap) to lure users into considering to involve themselves with a seemingly vulnerable contract to earn more funds. In case funds are not present, the users (or victims) have to motivation to transact with the contract.

3. A victim whose funds are stuck in the contract, this can be figured when the wallet address is different than the contract owner.

4. A transfer of profit from the contract to owner

5. A minimal presence of transfer of funds to different accounts (except the creator). Even if the transfer exists, the transferred output is usually significantly less that the invoking input.

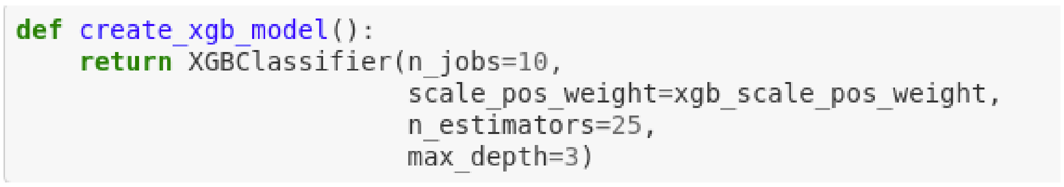
These can be described as a permutation with 8 variables, the combinations of which yields a total of 864 cases, out of which 244 can be considered as valid (practically possible):

Name	Possible Values
sender	creator, other
creation	yes, no
error	yes, no
balanceCreator	up, unchanged, down
balanceContract	up, unchanged, down
balanceSender	up, unchanged, down
balanceOtherPositive	yes, no
balanceOtherNegative	yes, no

Figure 2.6: Variables classified as Fund Flow

2.3.2.4 Training the Model

To perfect the model for classification, XGBoost [35, 36] is used as the machine learning model to deduce each contract into positive (honeypots) and negative (non-honeypots) class. The core function used for training the model on Google Cloud Instance is depicted as:



```python
def create_xgb_model():
    return XGBClassifier(n_jobs=10,
                         scale_pos_weight=xgb_scale_pos_weight,
                         n_estimators=25,
                         max_depth=3)
```

Figure 2.7: Classifier Function

Stratified k-fold cross-validation [37, 38] method with k=10 has been used to generalize the generated models for untrained, unseen and new data. This approach involves splitting the data into k subsets, which terminologically is referred to as folds. At a time, the model is trained on k-1 folds, while it is tested on one at a time. The stratified approach is used in particular to preserve class proportions.

The dataset also presents an imbalanced classification problem, since there are much more non-honeypot contracts on the Ethereum blockchain than honeypots i.e. the negative class is extremely skew with respect to the positive class which inherently makes the model biased towards the negative class. To counter this, the positive class i.e. the honeypot data in the XGBoost model is trained with an additional scaling weight to balance the data skewness.

On training the model with limited feature sets as identified in previous section, the importance of input variables from each category can be reduced, such as:

Category	Feature	Importance
All	fundFlowCase83	0.657
All	normalTransactionValueMean	0.107
All	numSourceCodeLines	0.071
Only Transactions	normalTransactionValueMean	0.576
Only Transactions	normalTransactionValueStd	0.117
Only Transactions	normalTransactionGasUsedStd	0.077
Only Source Code	numSourceCodeLines	0.424
Only Source Code	compilerPatchVersion136	0.129
Only Source Code	compilerPatchVersion125	0.070
Only Fund Flow	fundFlowCase83	0.799
Only Fund Flow	fundFlowCase201	0.036
Only Fund Flow	fundFlowCase79	0.032

Figure 2.8: Feature category-wise important features

2.4 Integration of Detection System

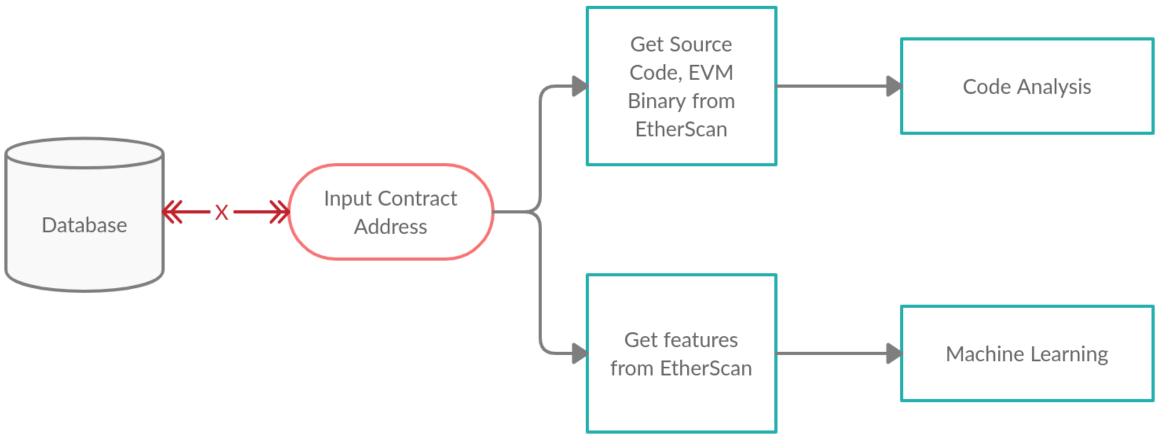

Database
X
Input Contract Address
Get Source Code, EVM Binary from EtherScan
Code Analysis
Get features from EtherScan
Machine Learning

Figure 2.9: Integration of Detection System

The Detection System is invoked only when the details of an existing contract is not found in database. In which case, inputs for both components (Code Analysis and Machine Learning Components) is fetched using EtherScan API and the result of honeypot is deduced by weighing in results from each component. It is worth noting that the Machine Learning model has given minimal false positives and hence it is weighed over the Code Analysis Component. This integration can be referred to in the attached block diagram.

2.5 Zero-Day Honeypot Detection

After training, it is evident that the Machine Learning Model is able to identify existing classified honeypots, likes of which were present in the training set. For testing the classification model against zero-day or unknown or new type of honeypots, the cross validation strategy is modified to consider one technique each time. The test set is created only with the unknown technique and the rest are considered as training set.

Table 2.1: Division of dataset for zero-day analysis

Test Set	**Training Set**
New Honeypot	All known honeypots, except the considered unknown (new) test honeypot

Since the test set only consists of unknown honeypot dataset, the efficiency of each technique directly provides the accuracy and ability of the model to detect zero-day honeypots. This efficiency and report are detailed in the Results section.

2.6 Database

A non-relational database MongoDb[39] is used for the purpose of identification of honeypots. There are 2 collections required for the purpose of this service. A MongoDb cluster with 2 additional replicas is setup on a Google Cloud instance. The following is a brief:

Table 2.2: Non-relational database schema

Collection	**Details**
Auth	Contains user details including _id (Unique user identifier), username (Username used to login), password (Hashed user password).
Contracts	Contains contract details including _id (Contract Address used as identifier), isHoneypot (Boolean classifying honeypot), category (Category of honeypot, if contract is honeypot and category is identifiable) and report (JSON report of respective honeypot).

2.7 REST API Service

The SaaS service can be consumed using a RESTful Web Service [40] which exposes multiple endpoints to interact with. Since this is a service, only authenticated users are allowed to access and user account access are currently granted manually as per requirement. Endpoints with an authentication requirement takes in a Bearer token[41] as the HTTP Request Header, which can be obtained via the /auth endpoint. The following is a brief API documentation:

Table 2.3: SaaS API Documentation

Endpoint	**Description**	**Auth**
POST /cryptoprobe/v1/auth	Takes username and password as input and returns an AUTHENTICATION token (JSON Web Token [42]) which is used to authenticate the user for further requests.	false
POST /cryptoprobe/v1/analyze	Takes a smart contract address as input and returns if a contract is honeypot or not. In case, the honeypot can be categorized, the appropriate category label is also returned.	true
POST /cryptoprobe/v1/details	Takes a smart contract address as input and returns all details available about a smart contract. This can be considered like a verbose mode which returns additional data for reports and debugging. If the analysis is not available for the provided contract, an error is returned.	true

This API can now be consumed via end-users, products, organizations, wallets, blockchains and individuals to integrate this service in their existing services and/or products, examples of which can be Web Applications, Wallets, Plugins, Extensions etc. to provide real-time and practical protection against honeypots.

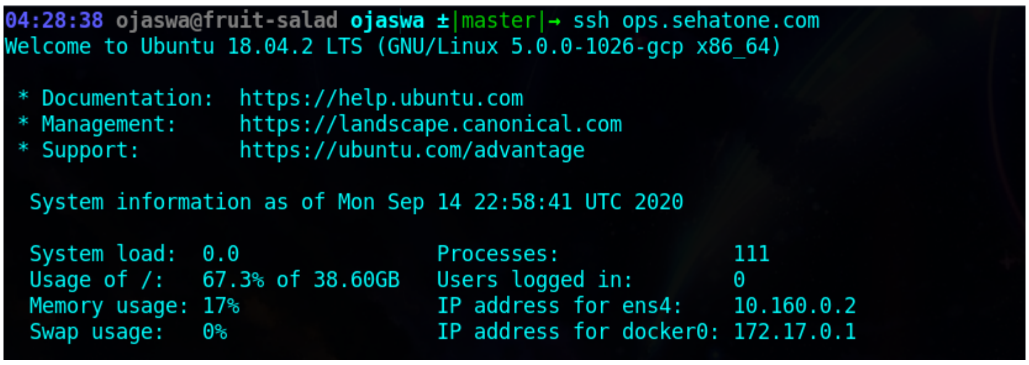
CHAPTER 3

Results and Discussions

This chapter highlights the satisfactory results obtained by the Service as a Service Platform ‘CryptoProbe‘ and discusses metrics and performance of various components of the project. It also discusses the deployment of the service and response in practical considerations.

3.1 Service

The complete service is containerized using advanced techniques such as ‘docker-compose‘ and deployed on personal Google Cloud and AWS virtual machines. The platform exposes an API interface allowing end-users to interact with the service and integrate it into their platforms.



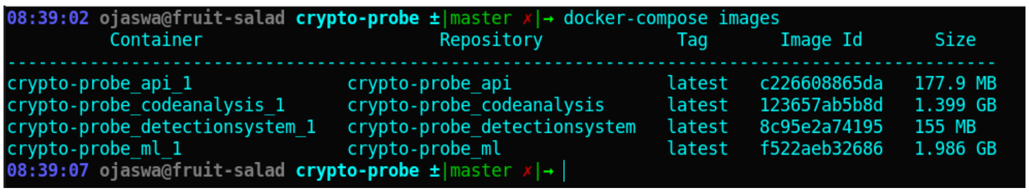
```
04:28:38 ojaswa@fruit-salad ojaswa ±|master|→ ssh ops.sehatone.com
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 5.0.0-1026-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Sep 14 22:58:41 UTC 2020

  System load:  0.0               Processes:             111
  Usage of /:   67.3% of 38.60GB  Users logged in:       0
  Memory usage: 17%               IP address for ens4:    10.160.0.2
  Swap usage:   0%                IP address for docker0: 172.17.0.1
```

Figure 3.1: Remote login to Virtual Machine via SSH

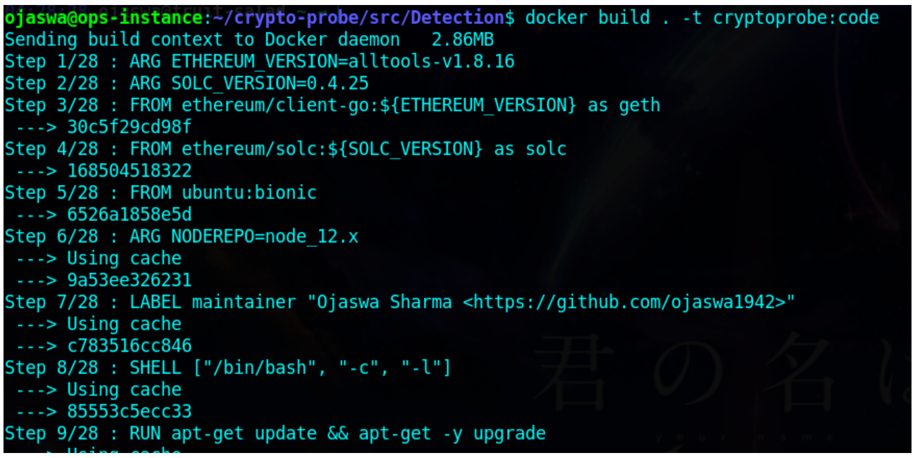


```
08:39:02 ojaswa@fruit-salad crypto-probe ±|master ✗|→ docker-compose images
          Container                       Repository              Tag       Image Id        Size
-----------------------------------------------------------------------------------------------------
crypto-probe_api_1                crypto-probe_api               latest   c226608865da   177.9 MB
crypto-probe_codeanalysis_1       crypto-probe_codeanalysis      latest   123657ab5b8d   1.399 GB
crypto-probe_detectionsystem_1    crypto-probe_detectionsystem   latest   8c95e2a74195   155 MB
crypto-probe_ml_1                 crypto-probe_ml                latest   f522aeb32686   1.986 GB
08:39:07 ojaswa@fruit-salad crypto-probe ±|master ✗|→ |
```

Figure 3.2: Containers running to facilitate detection and SaaS service

3.2 Detection System

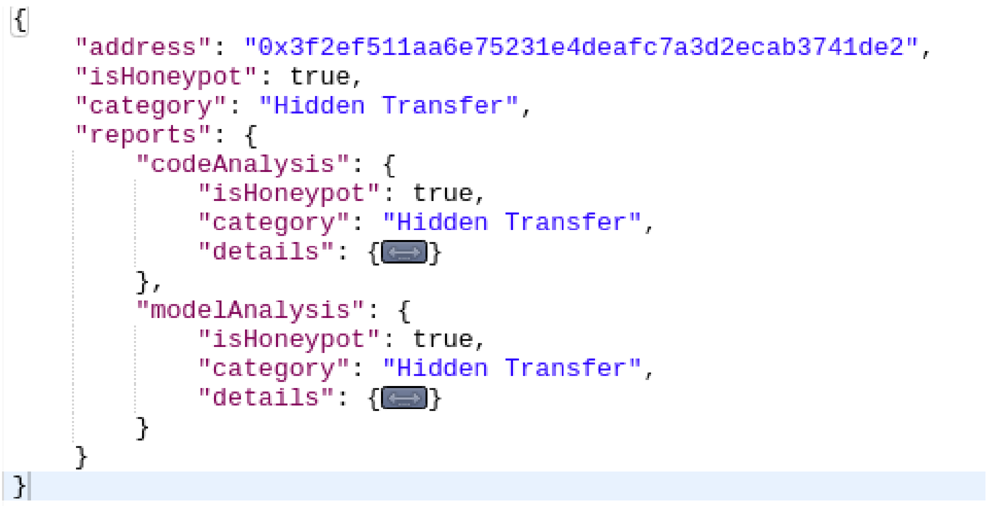
The honeypot detection system is the core of this service and is successfully integrated with 2 sub-components to provide unprecedented and weighed-in results from both of its components. The detection system is containerized by multiple docker[43] images and combined with each as service using docker-compose, which helps on-the-go deployment and scalability for the core of this service. The detection system consists of 3 docker images, one for each component, and another for its integration.



```
ojaswa@ops-instance:~/crypto-probe/src/Detection$ docker build . -t cryptoprobe:code
Sending build context to Docker daemon   2.86MB
Step 1/28 : ARG ETHEREUM_VERSION=alltools-v1.8.16
Step 2/28 : ARG SOLC_VERSION=0.4.25
Step 3/28 : FROM ethereum/client-go:${ETHEREUM_VERSION} as geth
 ---> 30c5f29cd98f
Step 4/28 : FROM ethereum/solc:${SOLC_VERSION} as solc
 ---> 168504518322
Step 5/28 : FROM ubuntu:bionic
 ---> 6526a1858e5d
Step 6/28 : ARG NODEREPO=node_12.x
 ---> Using cache
 ---> 9a53ee326231
Step 7/28 : LABEL maintainer "Ojaswa Sharma <https://github.com/ojaswa1942>"
 ---> Using cache
 ---> c783516cc846
Step 8/28 : SHELL ["/bin/bash", "-c", "-l"]
 ---> Using cache
 ---> 85553c5ecc33
Step 9/28 : RUN apt-get update && apt-get -y upgrade
```

Figure 3.3: Code Analysis: Build Docker Image

Figure 3.4 shows response sent by the detection system when a scan is initiated by the service API on user request, which reports that both components have deduced the contract address as a honeypot. This response is then stored in the database and exposed as required.

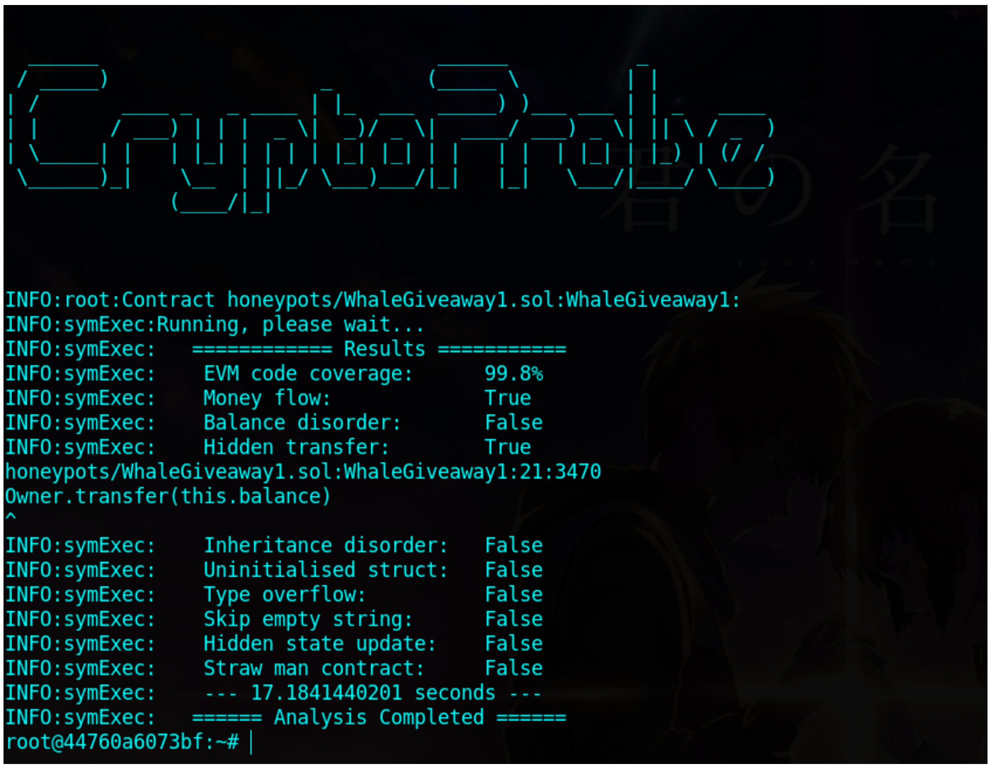


```
{
    "address": "0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2",
    "isHoneypot": true,
    "category": "Hidden Transfer",
    "reports": {
        "codeAnalysis": {
            "isHoneypot": true,
            "category": "Hidden Transfer",
            "details": {⟺}
        },
        "modelAnalysis": {
            "isHoneypot": true,
            "category": "Hidden Transfer",
            "details": {⟺}
        }
    }
}
```

Figure 3.4: Response from Detection System

When another request with same address is created, the detection system is not invoked. Since the contracts are immutable, this behaves as a permanent caching.

3.3 Detection System: Code Analysis



```
  ______                      ______       _
 / _____)             _      (_____ \     | |
| /       ____ _   _ ____ | |_  ___  _____) )___  ___ | |__   ____
| |      / ___) | | |  _ \|  _)/ _ \|  ____/ ___)/ _ \|  _ \ / _  )
| \_____| |   | |_| | | | | |_| |_| | |   | |   | |_| | |_) | (/ /
 \______)_|    \__  | ||_/ \___)___/|_|   |_|    \___/|____/ \____)
              (____/|_|


INFO:root:Contract honeypots/WhaleGiveaway1.sol:WhaleGiveaway1:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:      99.8%
INFO:symExec:     Money flow:             True
INFO:symExec:     Balance disorder:       False
INFO:symExec:     Hidden transfer:        True
honeypots/WhaleGiveaway1.sol:WhaleGiveaway1:21:3470
Owner.transfer(this.balance)
^
INFO:symExec:     Inheritance disorder:   False
INFO:symExec:     Uninitialised struct:   False
INFO:symExec:     Type overflow:          False
INFO:symExec:     Skip empty string:      False
INFO:symExec:     Hidden state update:    False
INFO:symExec:     Straw man contract:     False
INFO:symExec:     --- 17.1841440201 seconds ---
INFO:symExec:   ====== Analysis Completed ======
root@44760a6073bf:~# |
```

Figure 3.5: Code Analysis: Identified Honeypot with Hidden Transfer

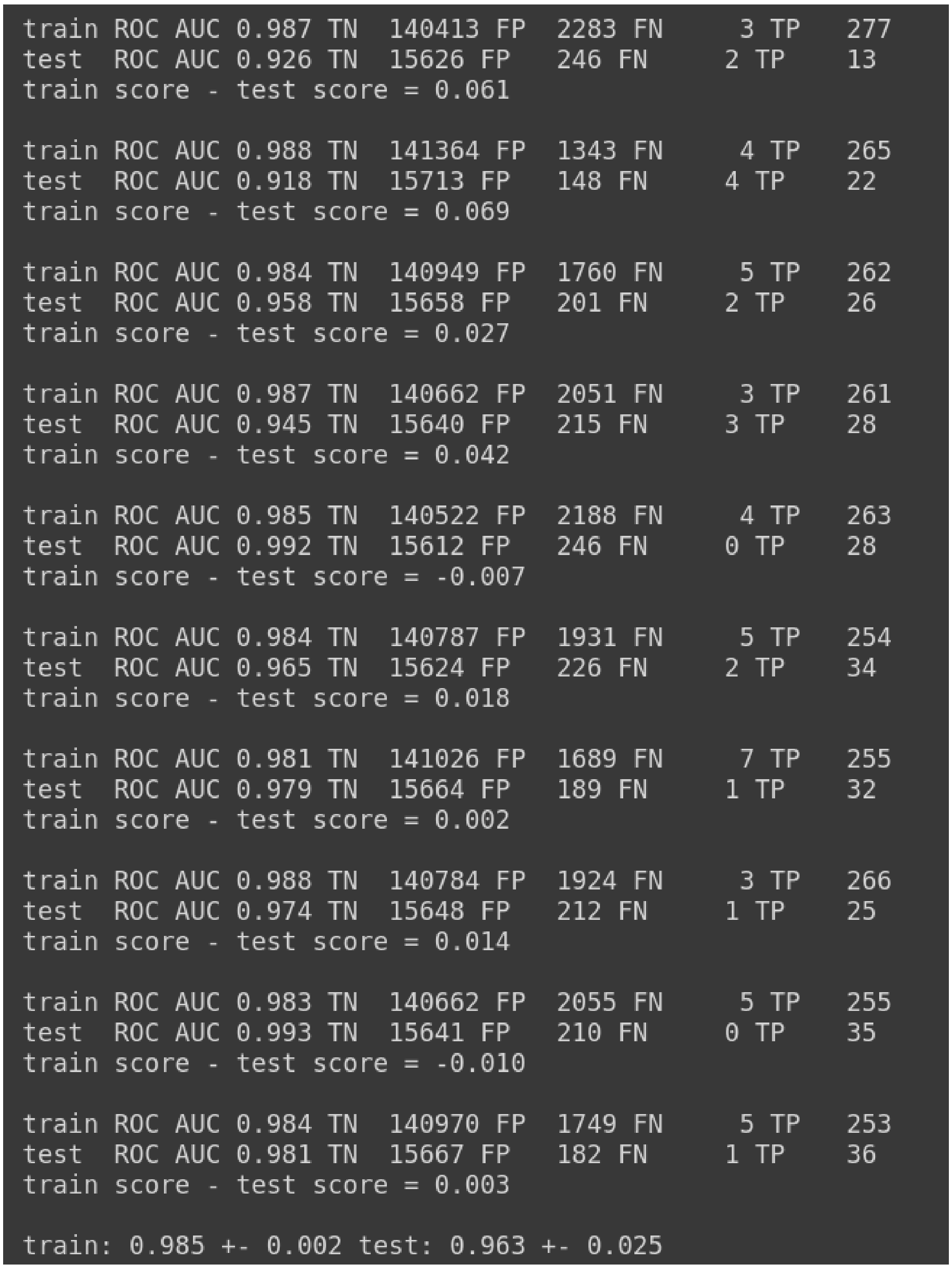
The code analysis section successfully identifies and categorizes honeypots according to studied taxonomy, by using symbolic execution paths and control flow graphs. The following are the configurations used:

Table 3.1: System Configurations used for Byte-Code Analysis in Detection System

Configuration	**Version**	**Requirement**
EVM	1.8.16	Disassembler
Solidity	0.4.25	Source (solidity) to byte-code compiler
Z3	4.7.1	Constraint Solver
Request Timeout	1 second	-
Global Contract Timeout	30 minutes	-
Loop Limit	10	-
DFS Limit	50	Max depth for control flow graphs
Gas Limit	4 Million	Computational fee allowance

3.4 Detection System: Machine Learning

To evaluate the effectiveness of created model, AUROC i.e. Area Under the Receiver Operating Characteristics [44] has been calculated to quantify the power, which portrays the True Positive Rate (TPR) against False Positive Rate (FPR) for various thresholds over the likelihood of classifying classes.



```
train ROC AUC 0.987 TN  140413 FP  2283 FN     3 TP   277
test  ROC AUC 0.926 TN  15626 FP   246 FN    2 TP    13
train score - test score = 0.061

train ROC AUC 0.988 TN  141364 FP  1343 FN     4 TP   265
test  ROC AUC 0.918 TN  15713 FP   148 FN    4 TP    22
train score - test score = 0.069

train ROC AUC 0.984 TN  140949 FP  1760 FN     5 TP   262
test  ROC AUC 0.958 TN  15658 FP   201 FN    2 TP    26
train score - test score = 0.027

train ROC AUC 0.987 TN  140662 FP  2051 FN     3 TP   261
test  ROC AUC 0.945 TN  15640 FP   215 FN    3 TP    28
train score - test score = 0.042

train ROC AUC 0.985 TN  140522 FP  2188 FN     4 TP   263
test  ROC AUC 0.992 TN  15612 FP   246 FN    0 TP    28
train score - test score = -0.007

train ROC AUC 0.984 TN  140787 FP  1931 FN     5 TP   254
test  ROC AUC 0.965 TN  15624 FP   226 FN    2 TP    34
train score - test score = 0.018

train ROC AUC 0.981 TN  141026 FP  1689 FN     7 TP   255
test  ROC AUC 0.979 TN  15664 FP   189 FN    1 TP    32
train score - test score = 0.002

train ROC AUC 0.988 TN  140784 FP  1924 FN     3 TP   266
test  ROC AUC 0.974 TN  15648 FP   212 FN    1 TP    25
train score - test score = 0.014

train ROC AUC 0.983 TN  140662 FP  2055 FN     5 TP   255
test  ROC AUC 0.993 TN  15641 FP   210 FN    0 TP    35
train score - test score = -0.010

train ROC AUC 0.984 TN  140970 FP  1749 FN     5 TP   253
test  ROC AUC 0.981 TN  15667 FP   182 FN    1 TP    36
train score - test score = 0.003

train: 0.985 +- 0.002 test: 0.963 +- 0.025
```

Figure 3.6: AUROC across each fold

AUROC is calculated and compared for both test and training for 4 feature sets, which include one feature set per source, and one with considering all of them together. Elaborating, the features are: Based on transactional data, based on source-code attributes, based on fund-flow frequencies and finally considering every single set collec-

tively. It is evident that the difference in efficiency between testing and training datasets is minute, which indicates how well the model responds to unseen and new data samples. Each of the above 4 experiment shows promising results, while using all sets together proves subtle advantage in results over others.
The following describes the final results collected across all folds:

Features	Train	Test
All	0.985 ± 0.002	0.968 ± 0.015
Only Transactions	0.966 ± 0.004	0.954 ± 0.030
Only Source Code	0.953 ± 0.002	0.942 ± 0.025
Only Fund Flow	0.952 ± 0.002	0.938 ± 0.023

Figure 3.7: Mean and Standard Deviation values for AUROC across each fold for test and training datasets

3.5 Detecting Zero-day Honeypots

As mentioned in the implementation section, to verify the efficiency of classifying unknown honeypots, the test dataset comprises of only the positive class (honeypots) belonging to the concerned unknown category. We can measure the test accuracy using the test Recall as measured using equation 3.1.

$$Recall = \frac{TP}{TP + FN} \tag{3.1}$$

The figure 3.8 shows False Negatives, True Positives and calculated Recall values for each type of honeypot contract, when it was removed from the training dataset. The high recall values for each case proves that the classification model is able to detect unknown, new, and hence zero-day honeypots.

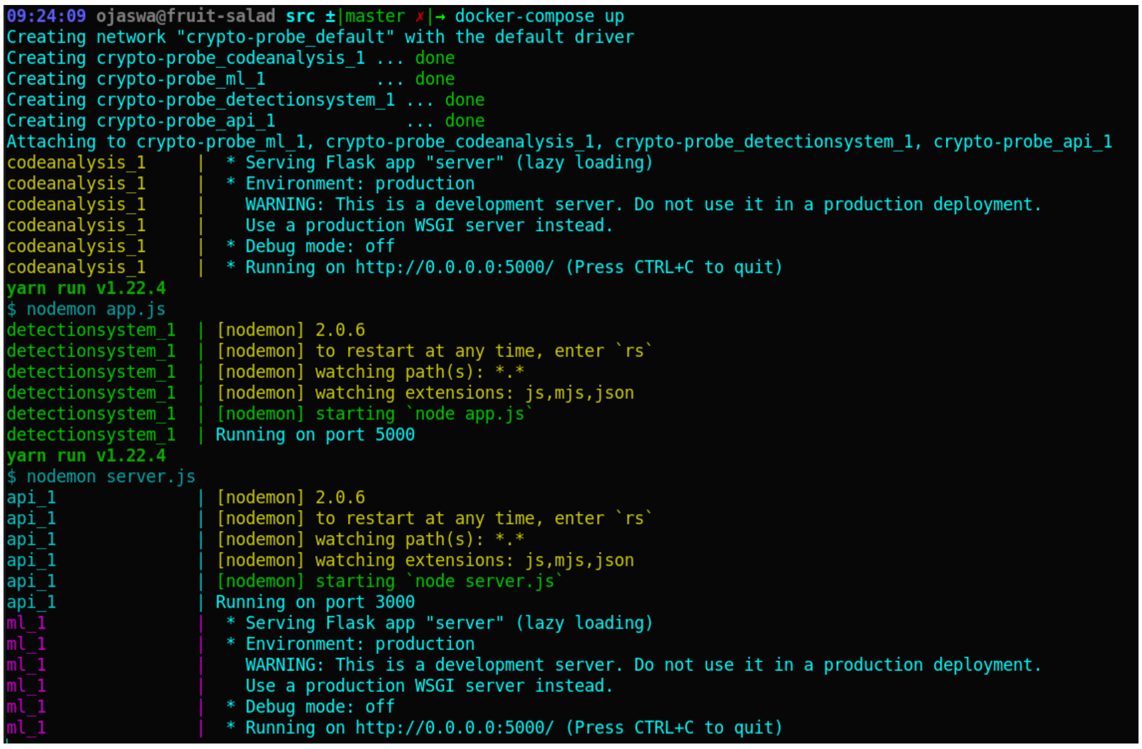
Removed Honeypot Technique	FN	TP	Recall
Type Deduction Overflow	0	4	1.000
Uninitialised Struct	2	37	0.949
Hidden Transfer	1	12	0.923
Hidden State Update	12	123	0.911
Inheritance Disorder	4	39	0.907
Straw Man Contract	3	28	0.903
Skip Empty String Literal	1	9	0.900
Balance Disorder	3	17	0.850

Figure 3.8: TP, FN and Recall values for various honeypot contracts when considered as unknown and excluded from training dataset

Furthermore, on using the trained model on the contract datasets, the model was able to uncover few new techniques of honeypots which would have been not possible to discover using the Code Analysis component without additionally manually intervening to add new flow conditions.

3.6 Service Deployment

The containerization of each individual component and their integrations establish that the service is instantly deployable and scalable as a single module or entity. With a single command, the system can be set up as required. Figure 3.9 shows how a single command is able to initiate all services as part of CryptoProbe, including the detection system and its sub-components.



```
09:24:09 ojaswa@fruit-salad src ±|master ✗|→ docker-compose up
Creating network "crypto-probe_default" with the default driver
Creating crypto-probe_codeanalysis_1 ... done
Creating crypto-probe_ml_1           ... done
Creating crypto-probe_detectionsystem_1 ... done
Creating crypto-probe_api_1             ... done
Attaching to crypto-probe_ml_1, crypto-probe_codeanalysis_1, crypto-probe_detectionsystem_1, crypto-probe_api_1
codeanalysis_1     |  * Serving Flask app "server" (lazy loading)
codeanalysis_1     |  * Environment: production
codeanalysis_1     |    WARNING: This is a development server. Do not use it in a production deployment.
codeanalysis_1     |    Use a production WSGI server instead.
codeanalysis_1     |  * Debug mode: off
codeanalysis_1     |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
yarn run v1.22.4
$ nodemon app.js
detectionsystem_1  | [nodemon] 2.0.6
detectionsystem_1  | [nodemon] to restart at any time, enter `rs`
detectionsystem_1  | [nodemon] watching path(s): *.*
detectionsystem_1  | [nodemon] watching extensions: js,mjs,json
detectionsystem_1  | [nodemon] starting `node app.js`
detectionsystem_1  | Running on port 5000
yarn run v1.22.4
$ nodemon server.js
api_1              | [nodemon] 2.0.6
api_1              | [nodemon] to restart at any time, enter `rs`
api_1              | [nodemon] watching path(s): *.*
api_1              | [nodemon] watching extensions: js,mjs,json
api_1              | [nodemon] starting `node server.js`
api_1              | Running on port 3000
ml_1               |  * Serving Flask app "server" (lazy loading)
ml_1               |  * Environment: production
ml_1               |    WARNING: This is a development server. Do not use it in a production deployment.
ml_1               |    Use a production WSGI server instead.
ml_1               |  * Debug mode: off
ml_1               |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Figure 3.9: Initiating docker-compose services to deploy CryptoProbe

It is also worth noting that each service has unique system requirements and technology stack but containers still make them possible to be run without separately establishing individual dependencies and setup. Error and processing logs are saved for each service separately, making the system easy to debug, develop and deploy.

CHAPTER 4

Conclusion

CryptoProbe - A Service as a Service platform to detect Honeypots on decentralized Ethereum blockchains using Code Analysis and Machine Learning was successfully developed and deployed.

1. Code Analysis technique using Control Flow Graphs (CFG) and Symbolic Link Execution has been implemented to identify known types of honeypots
2. Machine Learning is used to create a classification model to deduce a smart contract as honeypot using filtered data obtained from extensive feature extraction from contract data obtained via EtherScan API.
3. A classification model capable of detecting unknown and zero-day honeypots has been developed.
4. An overall system including multiple components such as Database, Detection System, Servers has been developed and deployed.
5. The created service has been exposed in the form of an API to allow users/developers to integrate the service into their systems and wallets for real-time protection against honeypots. A sample consumption of the exposed API has been also presented.
6. Future works include generalizing sub-components to support more versions, compilers, languages and blockchains to create an environment-agnostic system to detect honeypots. The Machine Learning component, in particular, can employ greater powered and efficient models using neural networks to increase efficiency and detection capabilities of the system.

REFERENCES

[1] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[2] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Manubot, Tech. Rep., 2019.

[3] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, “Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack,” *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[5] S. Palladino, “The parity wallet hack explained,” *July-2017.[Online]. Available: https://blog. zeppelin. solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7*, 2017.

[6] A. Sherbachev, “Hacking the hackers: Honeypots on ethereum network,” *Note: https://hackernoon. com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577 Cited by*, vol. 1, 2018.

[7] A. Sherbuck, “Dissecting an ethereum honeypot,” 2018.

[8] W. Chen, X. Guo, Z. Chen, Z. Zheng, Y. Lu, and Y. Li, “Honeypot contract risk warning on ethereum smart contracts,” in *2020 IEEE International Conference on Joint Cloud Computing*. IEEE, 2020, pp. 1–8.

[9] N. Szabo, “Smart contracts: building blocks for digital markets,” *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, 1996.

[10] L. Ethereum low-level lisp like, “language, january 2019,” https://lll-docs.readthedocs.io/en/latest/lllintroduction.html.

[11] vyperlang, “Vyper: Pythonic smart contract language for the evm,” https://github.com/vyperlang/vyper, 2019.

[12] CornellBlockchain, “Bamboo: A language for morphing smart contracts,” https://github.com/CornellBlockchain/bamboo, 2018.

[13] C. Dannen, *Introducing Ethereum and solidity.* Springer, 2017, vol. 1.

[14] E. Team, “Etherscan: The ethereum block explorer,” 2017.

[15] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International conference on principles of security and trust.* Springer, 2017, pp. 164–186.

[16] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, “Smart contracts vulnerabilities: a call for blockchain software engineering?” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE).* IEEE, 2018, pp. 19–25.

[17] A. Mense and M. Flatscher, “Security vulnerabilities in ethereum smart contracts,” in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, 2018, pp. 375–380.

[18] S. Zhou, M. Möser, Z. Yang, B. Adida, T. Holz, J. Xiang, S. Goldfeder, Y. Cao, M. Plattner, X. Qin *et al.*, “An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2793–2810.

[19] D. Perez and B. Livshits, “Smart contract vulnerabilities: Does anyone care?” *arXiv preprint arXiv:1902.06710*, 2019.

[20] C. F. Torres, M. Steichen *et al.*, “The art of the scam: Demystifying honeypots in ethereum smart contracts,” in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1591–1607.

[21] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[22] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[23] S. Khurshid, C. S. Păsăreanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2003, pp. 553–568.

[24] O. Coudert, C. Berthet, and J. C. Madre, “Verification of synchronous sequential machines based on symbolic execution,” in *International Conference on Computer Aided Verification*. Springer, 1989, pp. 365–373.

[25] C. Cadar and K. Sen, “Symbolic execution for software testing: three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[26] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[27] L. De Moura and N. Bjørner, “Satisfiability modulo theories: introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[28] S. Russell and P. Norvig, “Artificial intelligence: a modern approach,” 2002.

[29] L. Bilge and T. Dumitraş, “Before we knew it: an empirical study of zero-day attacks in the real world,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 833–844.

[30] K. Casey, “Software as a service (saas),” https://searchcloudcomputing.techtarget.com/definition/Software-as-a-Service.

[31] R. Camino, C. F. Torres, and R. State, “A data science approach for honeypot detection in ethereum,” *arXiv preprint arXiv:1910.01449*, 2019.

[32] R. Camino, C. F. Torres, M. Baden, and R. State, “A data science approach for detecting honeypots in ethereum,” in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020, pp. 1–9.

[33] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[34] melonproject, “Oyente - an analysis tool for smartcontracts v0.2.7 (commonwealth),,” https://github.com/melonproject/oyente, February 2017.

[35] T. Chen, T. He, M. Benesty, V. Khotilovich, and Y. Tang, “Xgboost: extreme gradient boosting,” *R package version 0.4-2*, pp. 1–4, 2015.

[36] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[37] J. D. Rodriguez, A. Perez, and J. A. Lozano, “Sensitivity analysis of k-fold cross validation in prediction error estimation,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 3, pp. 569–575, 2009.

[38] Y.-D. Zhang, Z.-J. Yang, H.-M. Lu, X.-X. Zhou, P. Phillips, Q.-M. Liu, and S.-H. Wang, "Facial emotion recognition based on biorthogonal wavelet entropy, fuzzy support vector machine, and stratified cross validation," *IEEE Access*, vol. 4, pp. 8375–8385, 2016.

[39] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[40] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[41] M. Jones and D. Hardt, "The oauth 2.0 authorization framework: Bearer token usage," RFC 6750, October, Tech. Rep., 2012.

[42] M. Jones, B. Campbell, and C. Mortimore, "Json web token (jwt) profile for oauth 2.0 client authentication and authorization grants," *May-2015.[Online]. Available: https://tools. ietf. org/html/rfc7523*, 2015.

[43] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[44] D. L. Streiner and J. Cairney, "What's under the roc? an introduction to receiver operating characteristics curves," *The Canadian Journal of Psychiatry*, vol. 52, no. 2, pp. 121–128, 2007.