

**CRYPTOPROBE: IDENTIFICATION & CURTAILMENT OF
HONEYPOTS ON DECENTRALIZED BLOCKCHAIN
NETWORKS USING MACHINE LEARNING AND CODE
ANALYSIS**

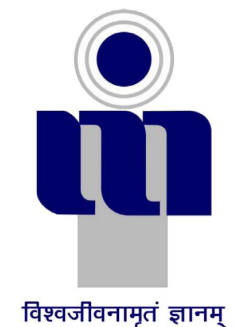
BTech Project Intermediate Report,
submitted in partial fulfillment of the requirement for the award of
Integrated Post Graduate (IPG) Master of Technology degree in
Information Technology

by

Ojaswa Sharma (2017IMT-102)

under the supervision of

Prof. Shashikala Tapaswi
Dr. K K Pattanaik



**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 September 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" - programs that are executed across a decentralised network of nodes which allow obligations consisting 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 trend towards a more proactive approach seems to be on the rise, where attackers do not search for vulnerable contracts anymore. Instead, they try to lure their victims into traps by deploying seemingly vulnerable contracts that, upon execution, unfold 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, 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

ABSTRACT	ii
LIST OF FIGURES	v
1 Introduction	1
1.1 Background	1
1.1.1 Smart Contracts	1
1.1.2 Ethereum Virtual Machine	2
1.1.3 Etherscan Blockchain Explorer	2
1.2 Motivation	3
1.3 Honeypots	3
1.4 Symbolic Execution	3
1.5 Data Science and Machine Learning	4
1.6 Zero-day Vulnerability	4
1.7 Service as a Service	5
2 Literature Survey	6
2.1 Security Bugs in Contracts	6
2.1.1 Transaction-Ordering Dependence	6
2.1.2 Timestamp Dependence	6
2.1.3 Mishandled Exceptions	7
2.1.4 Reentrancy Vulnerability	7
2.2 Taxonomy of Honeypots	7
2.2.1 Balance Disorder (BD)	7
2.2.2 Inheritance Disorder (ID)	8
2.2.3 Skip Empty String Literal (SESL)	8
2.2.4 Type Deduction Overflow (TDO)	8
2.2.5 Uninitialised Struct (US)	8
2.2.6 Hidden State Update (HSU)	8
2.2.7 Hidden Transfer (HT)	8
2.2.8 Straw Man Contract (SMC)	9

2.3	Honeypot Feature Extraction	9
3	Thesis Objectives and Deliverables	10
3.1	Objectives and Deliverables	10
4	System Architecture and Methodology	11
4.1	Phases of Honeypot	11
4.2	System Design	12
4.3	Detection System	12
4.3.1	Code Analysis	13
4.3.2	Machine Learning	14
4.3.2.1	Data Acquisition	14
4.3.2.2	Labels	14
4.3.2.3	Feature Detection	14
5	Implementation	15
5.1	Code Analysis	15
5.1.1	Symbolic Analysis	15
5.1.2	Cash Flow Analysis	15
5.1.3	Honeypot Analysis	16
5.2	Infrastructure	16
6	Results	17
6.1	Screenshots	17
7	Future Work	19
7.1	Tasks to be completed	19
	References	19

LIST OF FIGURES

4.1	Actors and phases of honeypot	11
4.2	Overall architecture of the system	12
4.3	High-level view of Detection System	12
4.4	An overview of analysis pipeline of the <i>Code Detection</i> component . . .	13
4.5	Architecture of Symbolic Analysis in The Oyente Tool[1]	13
5.1	Remote login to Virtual Machine via SSH	16
6.1	Build Docker Image	17
6.2	Run Docker Image	17
6.3	Identified Honeypot with Hidden Transfer	18
6.4	Identified Honeypot with Inheritance Disorder	18

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
BD	Balance Disorder
ID	Inheritance Disorder
ESL	Skip Empty String Literal
TDO	Type Deduction Overflow
US	Uninitialized Struct
HSU	Hidden State Update
SMC	Straw Man Contract
API	Application Programming Interface
CFG	Control Flow Graph
SHA	Secure Hash Algorithms
SSH	Secure Shell
DFS	Depth First Search

NOTATIONS

σ	World state
σ'	Modified world state
$\sigma[a]$	State of account a
$\sigma[a]b$	Balance of account or contract a
$\sigma[a]s$	Storage of contract a
I	Transaction execution environment
T_i, T_j	Transactions
I_a	Address of smart contract being executed
I_d	Transaction input data
I_v	Transaction value
I_s	Transaction sender

CHAPTER 1

Introduction

The concept of blockchain was introduced in 2009 with the release of Satoshi Nakamoto's Bitcoin [2], which has evolved greatly since then. It is regarded as one of the most disruptive technologies since the invention of the Internet itself. Modern blockchains such as Ethereum [3] aim to decentralise transactions and programs as a whole through so-called smart contracts which are deployed, invoked and removed from the blockchain via transactions. Ethereum has faced several devastating attacks [4, 5] on vulnerable smart contracts such as The DAO hack in 2016 and the Parity Wallet hack in 2017. Malicious contracts that look vulnerable but are exploitative are a rising trend. Recently, cyber attackers and criminals have started using these contracts to beguile users by deploying contracts that pretend to give away funds or services, but in fact contain hidden traps. These types of attacks were then named by the community as "Honeypots" [6, 7, 8].

1.1 Background

1.1.1 Smart Contracts

The notion of smart contracts has been introduced by Nick Szabo in 1997 [9]. He described the concept of a trustless system consisting of self-executing computer programs that would facilitate the digital verification and enforcement of contract clauses contained in legal contracts. However, this concept only became a reality with the release of Ethereum in 2015. Ethereum smart contracts are different from traditional programs in several aspects. For example, as the code is stored on the blockchain, it becomes immutable and its execution is guaranteed by the blockchain. Nevertheless, smart contracts may be destroyed, if they contain the necessary code to handle their destruction. Once destroyed, a contract can no longer be invoked and its funds are transferred to another address. Smart contracts are usually developed using a dedicated high-level programming language that compiles into low-level byte-code. The byte-

code of a smart contract is then deployed to the blockchain through a transaction. Once successfully deployed, a smart contract is identified by a 160-bit address. Despite a large variety of programming languages (e.g. Vyper [10], LLL[11] and Bamboo[12]), Solidity[13] remains the most prominent programming language for developing smart contracts in Ethereum. Solidity's syntax resembles a mixture of C and JavaScript. It comes with a multitude of unique concepts that are specific to smart contracts, such as the transfer of funds or the capability to call other contracts.

1.1.2 Ethereum Virtual Machine

The Ethereum blockchain consists of a network of mutually distrustful nodes that together form a decentralised public ledger. This ledger allows users to create and invoke smart contracts by submitting transactions to the network. These transactions are processed by so-called miners. Miners execute smart contracts during the verification of blocks, using a dedicated virtual machine denoted as the Ethereum Virtual Machine [3]. The EVM is a stack-based, register-less virtual machine, running low-level bytecode, that is represented by an instruction set of opcodes. To guarantee the termination of a contract and thus prevent miners from being stuck in endless loops of execution, the concept of gas has been introduced. It associates costs to the execution of every single instruction. When issuing a transaction, the sender has to specify the amount of gas that he or she is willing to pay to the miner for the execution of the smart contract. The execution of a smart contract results in a modification of the world state σ , a data structure stored on the blockchain mapping an address to an account state $\sigma[a]$. The account state of a smart contract consists of two main parts: a balance $\sigma[a]b$, which holds the amount of ether owned by the contract, and storage $\sigma[a]s$, which holds the persistent data of the contract. The EVM can essentially be seen as a transaction-based state machine that takes as input σ and I , and outputs a modified world state σ' .

1.1.3 Etherscan Blockchain Explorer

Etherscan [14] is an online platform that collects and displays blockchain specific information. It acts as a blockchain navigator allowing users to easily look up the contents of individual blocks, transactions and smart contracts on Ethereum. It offers multiple services on top of its exploring capabilities. One of these services is the possibility for smart contract creators to publish their source code and confirm that the bytecode stored under a specific address is the result of compilation of the specified source code. It also offers users the possibility to leave comments on smart contracts.

1.2 Motivation

A blockchain-based smart contract is visible to all users of said blockchain. However, this leads to a situation where bugs, including security holes, are visible to all yet may not be quickly fixed. Issues in Ethereum smart contracts, in particular, include ambiguities and easy-but-insecure constructs in its contract language Solidity, compiler bugs, Ethereum Virtual Machine bugs, attacks on the blockchain network, the immutability of bugs and that there is no central source documenting known vulnerabilities, attacks[15] and problematic constructs. Malicious contracts (honeypots) share one trait in common: they almost always try to look like they were designed by a beginner. As such, they are a great place to learn about some of the pitfalls that can befall a new entrant to the space, and serve as an interesting case study into the wild-west world of smart contract security. By analyzing a few of the more interesting cases of not-so-vulnerable contracts, we can gain a deeper understanding of how smart contract security works in practice.

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.

1.3 Honeypots

Definition (Honeypot)[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.*

In general, honeypots use a bait in the form of funds to lure users into traps. After looking at the source code, users are tricked into believing that there is a vulnerability and that they will receive more funds than they require to trigger the vulnerability. However, it is not clear to the users that the honeypot contains a hidden trapdoor that causes all or most of the funds to remain in the honeypot.

1.4 Symbolic Execution

Symbolic execution is a popular program analysis technique introduced in the mid '70s to test whether certain properties can be violated by a piece of software [21, 22, 23, 24,

25]. Aspects of interest could be that no division by zero is ever performed, no NULL pointer is ever dereferenced, no backdoor exists that can bypass authentication, etc. While in general there is no automated way to decide some properties (e.g., the target of an indirect jump), heuristics and approximate analyses can prove useful in practice in a variety of settings, including mission-critical and security applications.

Execution is performed by *symbolic* execution engine, which maintains for each explored control flow path: (i) a first-order Boolean formula that describes the conditions satisfied by the branches taken along that path, and (ii) a *symbolic* memory store that maps variables to symbolic expressions or values. Branch execution updates the formula, while assignments update the symbolic store. A model checker, typically based on a satisfiability modulo theories (SMT) solver [26, 27], is eventually used to verify whether there are any violations of the property along each explored path and if the path itself is realizable, i.e., if its formula can be satisfied by some assignment of concrete values to the program’s symbolic arguments.

1.5 Data Science and Machine Learning

Data science is a discipline that draws techniques from different fields like mathematics, statistics, computer science, and information science, with the goal of discovering insights from real world data. The term is closely related to other fields such as machine learning, data mining and business intelligence. Machine Learning is the sub-field of Artificial Intelligence that provides computer programs with the ability to improve from experience without being explicitly programmed [28]. Supervised machine learning algorithms can identify patterns that relate *features* (measurable characteristics of the data) to *labels* (a particular property of the data). The *learning* or training process takes place when the algorithms search for patterns based on examples for which the labels are known. The result is a model, an approximation of the underlying relationship between features and labels. During the *testing* step, the model assigns labels to samples that were not part of the training phase, and these assignments or *predictions* are compared to known labels. The purpose is to evaluate how well the model generalizes to unseen cases. If each label represents one option from a set of possible classes, a supervised machine learning problem is known as a *classification*.

1.6 Zero-day Vulnerability

The term “zero-day” refers to a newly discovered software vulnerability. Because the developer has just learned of the flaw, it also means an official patch or update to fix the issue hasn’t been released. So, “zero-day” refers to the fact that the developers

have “zero days” to fix the problem that has just been exposed — and perhaps already exploited by hackers. Once the vulnerability becomes publicly known, the vendor has to work quickly to fix the issue to protect its users. But the software vendor may fail[29] to release a patch before hackers manage to exploit the security hole. That’s known as a **zero-day attack**.

1.7 Service as a Service

Software as a service (SaaS) [30] is a software distribution model in which a third-party provider hosts applications and makes them available to customers over the Internet. SaaS is one of three main categories of cloud computing, alongside infrastructure as a service and platform as a service (IaaS and PaaS). SaaS is closely related to the application service provider (ASP) and on demand computing software delivery models. The hosted application management model of SaaS is similar to ASP, where the provider hosts the customer’s software and delivers it to approved end users over the internet. In the software on demand SaaS model, the provider gives customers network-based access to a single copy of an application that the provider created specifically for SaaS distribution. The application’s source code is the same for all customers and when new features or functionalities are rolled out, they are rolled out to all customers.

CHAPTER 2

Literature Survey

2.1 Security Bugs in Contracts

L Luu et al. [1] investigate the security of running smart contracts based on Ethereum in an open distributed network and introduce several new security problems in which an adversary can manipulate smart contract execution to gain profit. The studies suggests existence of subtle gaps in the understanding of the distributed semantics of the underlying platform.

2.1.1 Transaction-Ordering Dependence

A block includes a set of transactions, hence the blockchain state is updated several times in each epoch. Let us consider a scenario where the blockchain is at state σ and the new block includes two transactions (e.g., T_i, T_j) invoking the same contract. In such a scenario, users have uncertain knowledge of which state the contract is at when their individual invocation is executed. Only the miner who mines the block can decide the order of these transactions, consequently the order of updates. As a result, the final state of a contract depends on how the miner orders the transactions invoking it. We call such contracts as transaction-ordering dependent (or TOD) contracts.

2.1.2 Timestamp Dependence

A contract may use the block timestamp as a triggering condition to execute some critical operations, e.g., sending money. When mining a block, a miner has to set the timestamp for the block. Normally, the time-stamp is set as the current time of the miner's local system. However, the miner can vary this value by roughly 900 seconds, while still having other miners accept the block. Thus, the adversary can choose different block timestamps to manipulate the outcome of timestamp-dependent contracts.

2.1.3 Mishandled Exceptions

In Ethereum, there are several ways for a contract to call another, e.g., via send instruction or call a contract's function directly (e.g., `aContract.someFunction()`). If there is an exception raised (e.g., not enough gas, exceeding call stack limit) in the callee contract, the callee contract terminates, reverts its state and returns false. However, depending on how the call is made, the exception in the callee contract may or may not get propagated to the caller. For example, if the call is made via the send instruction, the caller should explicitly check the return value to verify if the call has been executed properly. This inconsistent exception propagation policy leads to many cases where exceptions are not handled properly. Nearly 27.9% of the contracts do not check the return values after calling other contracts via send.

2.1.4 Reentrancy Vulnerability

Reentrancy is a well-known vulnerability with the recent The Dao hack [4], where the attacker exploited the reentrancy vulnerability to steal over 3,600,000 Ether, or 60 million US Dollars at the time the attack happened. In Ethereum, when a contract calls another, the current execution waits for the call to finish. This can lead to an issue when the recipient of the call makes use of the intermediate state the caller is in. This may not be immediately obvious when writing the contract if possible malicious behavior on the side of the callee is not considered.

2.2 Taxonomy of Honeypots

Torres et al. [20] identified eight different honeypot techniques. Honeypots use a bait in the form of funds to lure users into traps. In the following, they provide a summary of the hidden trapdoors that honeypots use to deceive naive users.

2.2.1 Balance Disorder (BD)

This honeypot technique uses the fact that some users are misinformed regarding the exact moment at which a contract's balance is updated. A honeypot using this technique promises to return its balance and the value sent to it, if the latter is larger than the balance. However, the sent value is added to the balance before executing the honeypot's code. As a result, the balance will never be smaller than the sent value.

2.2.2 Inheritance Disorder (ID)

This honeypot technique involves two contracts and an inheritance relationship between them. Both the parent and the child contract each define a variable with the same name that guards the access to the withdrawal of funds. Although novice users may believe that both variables are the same, they are treated differently by the smart contract. As a result, changing the content of one variable does not change the content of the other variable.

2.2.3 Skip Empty String Literal (SESL)

A bug previously contained in the the Solidity compiler skips hard coded empty string literals in the parameters of function calls. Thus, the parameters following the empty string literal are shifted up-wards. This can be used to redirect the transfer of funds to the attacker instead of the users.

2.2.4 Type Deduction Overflow (TDO)

Although variables are usually declared with specific types, type deduction is also supported. A type is then inferred, without the guarantee that it is sufficiently large. This can be used to cause integer overflows that prematurely terminate execution and make the honeypot deviate from the behavior expected by the user.

2.2.5 Uninitialised Struct (US)

Structs are a convenient way in Solidity to group information. However, they need to be initialised appropriately, as values written to an uninitialised struct are written by default to the beginning of the storage. This overwrites existing values and modifies the contract's state. Thus, variables required to perform the transfer of funds may be overwritten and thereby prevent the transfer.

2.2.6 Hidden State Update (HSU)

Etherscan does not display internal transactions that do not transfer any value. This can be used to hide state updates from users and therefore make them believe that the contract is in a totally different state.

2.2.7 Hidden Transfer (HT)

Platforms such as Etherscan allow users to view the source code of published smart contracts. However, due to the limited size of the source code window, publishers can

insert white spaces to hide the displaying of certain source code lines by pushing them outside the window. This allows publishers to inject and hide code that transfers funds to them instead of users.

2.2.8 Straw Man Contract (SMC)

When uploading a contract's source code to Etherscan, only the contract itself is verified and not the code of contracts that the contract may call. For example, a source code file may contain two contracts, where the first contract makes calls to the second contract. The second one however, is actually not used after deployment. Instead, it is a straw man standing in place of another contract that selectively reverts transactions that transfer funds to users.

2.3 Honeyplot Feature Extraction

Camino et al. [31, 32] create a set of features in order to obtain a better understanding of the data, to verify that the behavior of transactions reflects the used honeypot technique. The features are divided into three categories, depending on where they originate from:

- a. Source Code Information
- b. Transaction Properties
- c. Fund Flows

The features are based on some assumptions (verified) and behaviors observed after considering a huge set of data. The events related to fund movements were studied, in order to verify if honeypots behave as we expect. For this, they query data by partially defining some of the fund flow variables and adding the frequencies of all matching cases.

CHAPTER 3

Thesis Objectives and Deliverables

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

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 4

System Architecture and Methodology

4.1 Phases of Honeypot

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

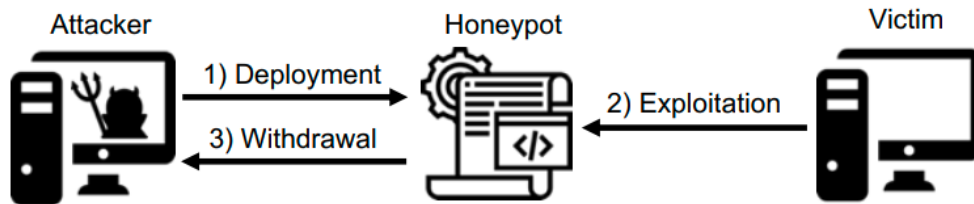


Figure 4.1: Actors and phases of honeypot

Figure 4.1 depicts the different actors and phases of a honey-pot. A honeypot generally operates in three phases:

1. The attacker deploys a seemingly vulnerable contract and places a bait in the form of funds;
2. The victim attempts to exploit the contract by transfer-ring at least the required amount of funds and fails;
3. The attacker withdraws the bait together with the funds that the victim lost in the attempt of exploitation.

An attacker does not require special capabilities to set up a honeypot. In fact, an attacker has the same capabilities as a regular Ethereum user. He or she solely requires the necessary funds to deploy the smart contract and place a bait.

4.2 System Design

The project can be viewed as 2 independent components: The **Detection System** and the **SaaS System**, which leverages the first. The overall system architecture is well depicted in Figure 4.2.

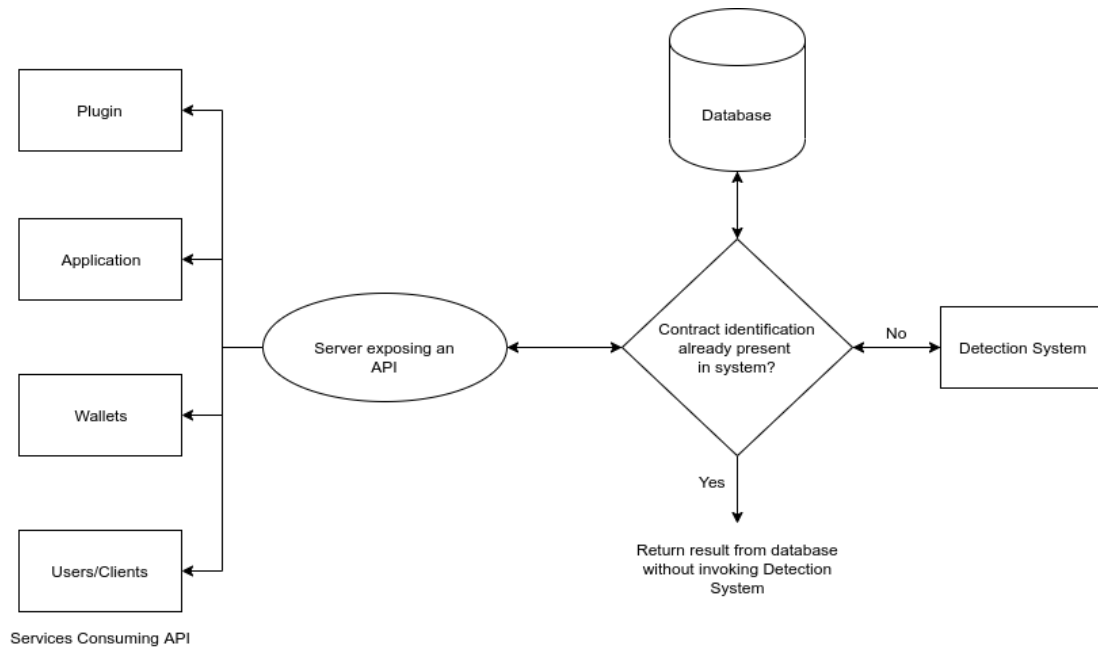


Figure 4.2: Overall architecture of the system

4.3 Detection System

The detection system contains two units, as represented in the Figure 4.3. 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.

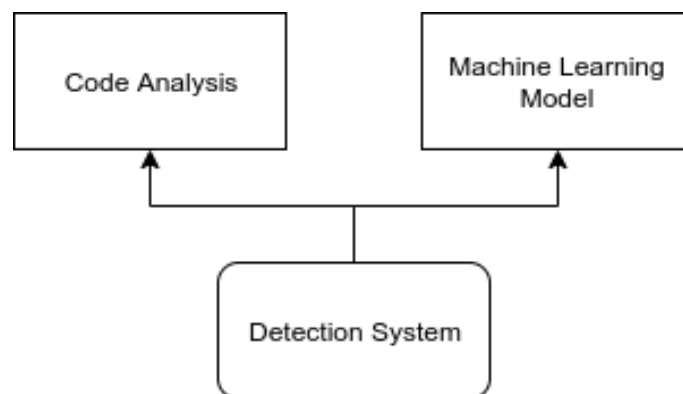


Figure 4.3: High-level view of Detection System

4.3.1 Code Analysis

Figure 4.4 depicts the overall architecture and analysis pipeline of the CodeAnalysis component. CodeAnalysis takes EVM bytecode as input and returns a detailed report regarding the different honeypot techniques it detected as output. It consists of three main components: **Symbolic Analysis**, **Cash Flow Analysis**, and **Honeypot Analysis**. The symbolic analysis component constructs the control flow graph (CFG) and symbolically executes its different paths. The result of the symbolic analysis is afterwards propagated to the cash flow analysis component as well as the honeypot analysis component. The cash flow analysis component uses the result of the symbolic analysis to detect whether the contract is capable to receive as well as transfer funds. Finally, the honeypot analysis component aims at detecting the different honeypots techniques studied using a combination of heuristics and the results of the symbolic analysis. Each of the three components uses the Z3 SMT [33] solver to check for the *satisfiability* of constraints.

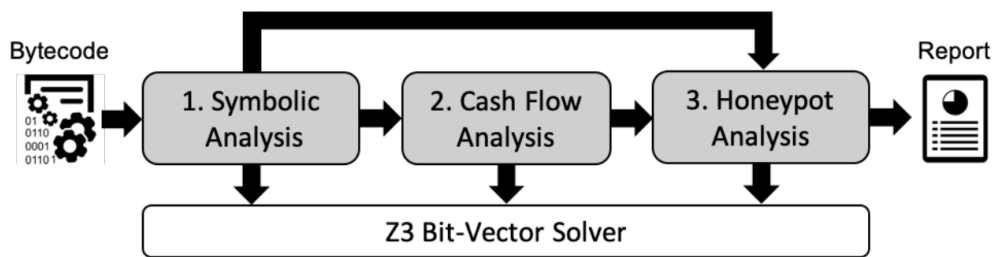


Figure 4.4: An overview of analysis pipeline of the *Code Detection* component

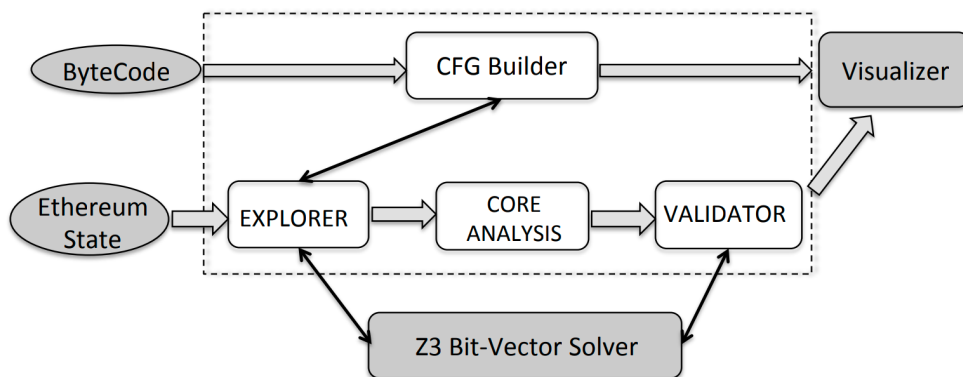


Figure 4.5: Architecture of Symbolic Analysis in The Oyente Tool[1]

The analysis tool is based upon symbolic execution [22, 1]. Symbolic execution represents the values of program variables as symbolic expressions of the input symbolic values. Each symbolic path has a path condition which is a formula over the

symbolic inputs built by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is infeasible if its path condition is unsatisfiable. Otherwise, the path is feasible. Symbolic execution is chosen because it can statically reason about a program path-by-path. As an example, to detect the transaction-ordering dependence (TOD), we must compare the outcomes of the interleaving of different execution paths. It is difficult to approach this with dynamic testing, given the non-determinism and complexity of the blockchain behaviors.

4.3.2 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.

4.3.2.1 Data Acquisition

This study is limited to roughly 2 million smart contracts created in the first blocks of the Ethereum blockchain. Using the Etherscan API [14], compilation and source code information for nearly 160,000 contracts was acquired, and around 141M normal and 4.6M internal transactions were downloaded.

4.3.2.2 Labels

The authors presenting taxonomy of honeypots (Section 2.2)[20] developed a tool that generated labels, and for each case they indicated if the labeling was correct after a manual inspection of the contract source code. For groups of honeypot contracts that share the exact same byte code, the authors only included one representative address in their list. In order to obtain a label for every contract, the SHA256 hash for the bytecode of each contract is calculated, and grouped on basis of the hashes. We then transfer the label of each representative contract to every contract in its group. We only label as honeypot the contracts that were manually confirmed by the authors, and ignore the false positives generated by their tool.

4.3.2.3 Feature Detection

Like mentioned in the previous sections, features are divided into 3 categories depending on where they originate from. These features will be used to implement the classification model.

CHAPTER 5

Implementation

5.1 Code Analysis

Code Analysis section of the overall service 'CryptoProbe' is implemented in Python with roughly 4,000 lines of code. Following is a brief description of implementation of each components:

5.1.1 Symbolic Analysis

The symbolic analysis component starts by constructing a CFG from the bytecode, where every node in the CFG corresponds to a basic block and every edge corresponds to a jump between individual basic blocks. A reused and modified version of the symbolic execution engine proposed by Luu et al. [32, 34] is used. The engine consists of an interpreter loop that receives a basic block and symbolically executes every single instruction within that block. The loop continues until all basic blocks of the CFG have been executed or a timeout is reached. Loops are terminated once they exceed a globally defined loop limit. The engine follows a depth first search approach when exploring branches and queries Z3 to determine their feasibility.

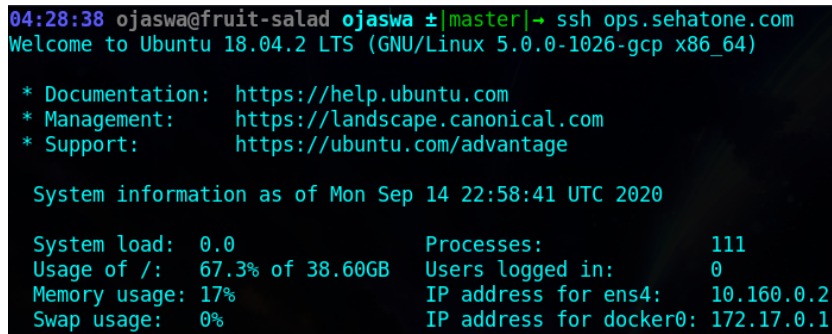
5.1.2 Cash Flow Analysis

A honeypot must be able to receive funds (e.g. the investment of a victim) and transfer funds (e.g. the loot of the attacker). The purpose of cash flow analysis is to improve the performance of the tool, by safely discarding contracts that cannot receive or transfer funds. This is identified by iterating all execution paths and checking whether there exists an execution path p , that does not terminate in a *REVERT*. If p satisfies the constraint $I_v > 0$, we know that funds can flow into the contract. Similarly, funds can flow out of the contract if we find at least one call c or execution path p that terminates in *SELFDESTRUCT* and satisfies constraints.

5.1.3 Honeypot Analysis

The honeypot analysis consist of several subsections, 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: We iterate over all calls contained in C and report a balance disorder, if we find a call c within an infeasible basic block, where $cv = Iv + \sigma[Ia]b$.

5.2 Infrastructure



```
04:28:38 ojuaswa@fruit-salad ojuaswa ±|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 5.1: Remote login to Virtual Machine via SSH

The current system architecture runs on a virtual machine on a Google Cloud instance facilitated by Docker[35] which is used to facilitate system-level isolation and containerization. We used version 1.8.16 of Geth’s EVM as disassembler and Solidity version 0.4.25 as source-code-to-bytecode compiler. As our constraint solver, Z3 version 4.7.1 was used. We set a timeout of 1 second per Z3 request for the symbolic execution. The symbolic execution’s global timeout is set to 30 minutes per contract. The loop limit, depth limit (for DFS) and gas limit for the symbolic execution are set to 10, 50 and 4 million, respectively.

CHAPTER 6

Results

The code analysis component of the proposed system *CryptoProbe* successfully identifies and classifies honeypots with respect to studied taxonomies.

6.1 Screenshots

```
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
--> Using cache
```

Figure 6.1: Build Docker Image

```
ojaswa@ops-instance:~/crypto-probe/src/Detection$ docker run -it cryptoprobe:code
root@44760a6073bf:~# python cryptoprobe/cryptoprobe.py -s honeypots/
CryptoRoulette.sol      GuessNumber.sol      PINCODE.sol          TestBank.sol          X2_FLASH.sol
DividendDistributor.bin ICO_Hold.sol          PrivateBank-Logger.bin TestToken.sol          firstTest.sol
DividendDistributor.sol IFKRYGE.sol           PrivateBank.sol       TransferReg.sol       testBank2.sol
EtherBet.sol            KingOfTheHill.sol     RACEFORETH.sol       TrustFund.sol
For Test.sol            MultiplicatorX3.sol   RichestTakeAll.sol    TwelHourTrains.sol
G_GAME.sol              NEW_YEARS_GIFT.sol    TerrionFund.sol       WhaleGiveaway1.sol
Gift_1_Eth.sol          OpenAddressLottery.sol Test1.sol              X2_FLASH.bin
root@44760a6073bf:~# python cryptoprobe/cryptoprobe.py -s honeypots/WhaleGiveaway1.sol
```

Figure 6.2: Run Docker Image

```

CrypToR0ad

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 6.3: Identified Honeypot with Hidden Transfer

```

CrypToR0ad

INFO:root:Contract honeypots/KingOfTheHill.sol:KingOfTheHill:
INFO:symExec:Running, please wait...
INFO:symExec: ===== Results =====
INFO:symExec:   EVM code coverage:      99.7%
INFO:symExec:   Money flow:                True
INFO:symExec:   Balance disorder:            False
INFO:symExec:   Hidden transfer:            False
INFO:symExec:   Inheritance disorder:      True
honeypots/KingOfTheHill.sol:KingOfTheHill:28:13
owner = msg.sender
^
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:   --- 1.67040705681 seconds ---
INFO:symExec:   ===== Analysis Completed =====

```

Figure 6.4: Identified Honeypot with Inheritance Disorder

CHAPTER 7

Future Work

While the current system is capable of detecting various types of honeypots, it cannot yet detect unknown and zero-day honeypots without manually adding conditions for symbolic execution.

7.1 Tasks to be completed

- i Preprocess data to understand potential data parameters and Machine Learning algorithms for Detection System. To experiment, determine and implement suitable algorithm and methodology for honeypot detection using data science.
- ii Merge individual components of the detection system i.e. code analysis and machine learning and perform an integrated analysis for robust and efficient results with minimal false positives.
- iii Design and develop service architecture for the SaaS solution. This includes integrating database for storing results and exposing the detection system as a function for server and API usage.
- iv Create a server and an API to handle & manage the SaaS solution. This step also incorporates consolidating various components (Server, Database, Detection System etc.) to realize the overall system architecture.
- v Deploy the system architecture with appropriate production measures e.g. containerization, load balancing, security etc.

If time and feasibility allows, we can also:

- vi Create a web application as a proof of concept depicting possible real-life usage of *CryptoProbe*, the SaaS platform for honeypot detection.

REFERENCES

- [1] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
- [2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.
- [3] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
- [4] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.
- [5] Santiago 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, 1, 2018.
- [7] Alex Sherbuck. Dissecting an ethereum honeypot, 2018.
- [8] Weili Chen, Xiongfeng Guo, Zhiguang Chen, Zibin Zheng, Yutong Lu, and Yin Li. Honeypot contract risk warning on ethereum smart contracts. In *2020 IEEE International Conference on Joint Cloud Computing*, pages 1–8. IEEE, 2020.
- [9] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, (16), 18(2), 1996.
- [10] vyperlang. Vyper: Pythonic smart contract language for the evm. <https://github.com/vyperlang/vyper>, 2019.

- [11] LLL Ethereum low-level lisp like. language, january 2019. <https://lll-docs.readthedocs.io/en/latest/lllinintroduction.html>.
- [12] CornellBlockchain. Bamboo: A language for morphing smart contracts. <https://github.com/CornellBlockchain/bamboo>, 2018.
- [13] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.
- [14] Etherscan Team. Etherscan: The ethereum block explorer, 2017.
- [15] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.
- [16] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE, 2018.
- [17] Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, pages 375–380, 2018.
- [18] Shunfan Zhou, Malte Möser, Zhemin Yang, Ben Adida, Thorsten Holz, Jie Xiang, Steven Goldfeder, Yinzhi Cao, Martin Plattner, Xiaojun 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)*, pages 2793–2810, 2020.
- [19] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710*, 2019.
- [20] Christof Ferreira Torres, Mathis Steichen, et al. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1591–1607, 2019.
- [21] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.
- [22] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

- [23] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.
- [24] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.
- [25] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
- [26] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
- [27] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
- [28] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.
- [29] Leyla Bilge and Tudor 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*, pages 833–844, 2012.
- [30] Kathleen Casey. Software as a service (saas). <https://searchcloudcomputing.techtarget.com/definition/Software-as-a-Service>.
- [31] Ramiro Camino, Christof Ferreira Torres, and Radu State. A data science approach for honeypot detection in ethereum. *arXiv preprint arXiv:1910.01449*, 2019.
- [32] Ramiro Camino, Christof Ferreira Torres, Mathis Baden, and Radu State. A data science approach for detecting honeypots in ethereum. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.
- [33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [34] melonproject. Oyente - an analysis tool for smartcontracts v0.2.7 (commonwealth),. <https://github.com/melonproject/oyente>, February 2017.
- [35] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.