The Greed Trap: Uncovering Intrinsic Ethereum Honeypots through Symbolic Execution

by

Mahtab NOROUZI

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS
M.A.Sc.

MONTREAL, APRIL 24, 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Kaiwen Zhang, Thesis supervisor
Department of Software Engineering and IT, École de technologie supérieure

Mr. Ali Ouni, Chair, Board of Examiners
Department of Software Engineering and IT, École de technologie supérieure

Mr. Fabio Petrillo, External Examiner
Department of Software Engineering and IT, École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON APRIL 18, 2023

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

Firstly, I would like to express my gratitude to my supervisor, Professor Kaiwen Zhang, for his continuous support, guidance, and encouragement throughout my research. His expertise and insights have been invaluable in shaping the direction and focus of my work. I am deeply appreciative of his kindness and unwavering support, which has inspired me to always put my best efforts into the project.

I am also grateful to my friend and collaborator, Mounir, whose incredible support made it possible to overcome numerous challenges at every step of this project. I want to extend my heartfelt thanks to my family members, my mother Zohre, my father Mehdi, my sister Minoo, and my grandmother, Forough whose unwavering love and encouragement have been a constant source of inspiration for me.

Lastly, I would like to take a moment to acknowledge the incredible courage and resilience of the women in Iran. Their bravery over the last few months has been truly inspiring to me and has served as a reminder of the importance of standing up for what I believe in. I stand with them wholeheartedly in the pursuit of a brighter future and the ultimate goal of achieving Women, Life, Freedom.

Le Piège de la cupidité : Découverte des honeypots intrinsèques à Ethereum grâce à l'exécution symbolique

Mahtab NOROUZI

RÉSUMÉ

Les contrats intelligents sont des programmes informatiques auto-exécutables qui s'exécutent sur des réseaux de blockchain et facilitent les transactions sécurisées, transparentes et décentralisées. La sécurité des contrats intelligents a toujours été une question critique dans la communauté de la blockchain, et l'une des principales préoccupations ces dernières années est la prévalence des honeypots - des contrats malveillants conçus pour tromper les utilisateurs en leur faisant déposer des fonds, pour découvrir qu'ils sont incapables de retirer leur argent et ont perdu leur dépôt initial. Dans cette recherche, nous présentons une nouvelle classification de honeypots et introduisons un nouveau type de contrat qui permet le développement d'une méthode future-proof pour détecter les honeypots basée sur le propriétaire du contrat et le flux de trésorerie. Nous mettons en œuvre cette méthode sous la forme d'un outil de détection appelé HoneyVader en utilisant une exécution symbolique pour identifier les contrats honeypot du monde réel. Nous appliquons notre outil à plus de 2 millions de contrats déployés sur le réseau Ethereum et détectons 139 honeypots. En utilisant cet outil, nous découvrons des honeypots zero-day et de nouvelles techniques utilisées par les attaquants, en plus de celles identifiées dans les travaux précédents.

Mots-clés: exécution symbolique, Ethereum, honeypot

The Greed Trap: Uncovering Intrinsic Ethereum Honeypots through Symbolic Execution

Mahtab NOROUZI

ABSTRACT

Smart contracts are self-executing computer programs that run on blockchain networks and facilitate secure, transparent, decentralized transactions. The security of smart contracts has always been a critical issue in the blockchain community, and one of the major concerns in recent years is the prevalence of honeypots – malicious contracts designed to deceive users into depositing funds, only to find that they are unable to withdraw their money and have lost their original deposit. In this research, we present a novel classification of honeypots and introduce a new type of contract that enables the development of a future-proof method for detecting honeypots based on the contract owner and cash flow. We implement this method in the form of a detection tool called HoneyVader using symbolic execution to identify real-world honeypot contracts. We apply our tool to over 2 million contracts deployed on the Ethereum network and detect 139 honeypots. Using this tool, we uncover zero-day honeypots and new techniques used by attackers, in addition to the ones identified in previous works.

Keywords: symbolic execution, Ethereum, honeypots

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LISTINGS

LIST OF ABBREVIATIONS

BTC	Bitcoin Cryptocurrency
DAO	Decentralized Autonomous Organization
DLT	Distributed Ledger Technology
PoW	Proof of Work
PoS	Proof of Stake
EVM	Ethereum Virtual Machine
ETH	Ether
EOA	Externally Owned Account
CA	Contract Account
API	Application Programming Interface
IOT	Internet Of Things
ML	Money Laundering
HYIP	High Yield Investment Programs
P&D	Pump and Dump
DoS	Denial of Service
CF	Cash Flow
DeFi	Decentralized Finance
dApp	Decentralized Application
UOT	Unique Owner Transfer

RPC	Remote Procedure Call
BD	Balance Disorder
ID	Inheritance Disorder
SESL	Skip Empty String Literal
TDO	Type Deduction Overflow
US	Uninitialized Struct
HT	Hidden Transfer
RT	Racing Time
MKET	Map Key Encoding Trick
TP	True Positive
FP	False Positive
FN	False Negative
IDE	Integrated Development Environment
BSC	Binance Smart Chain
URL	Uniform Resource Locator

INTRODUCTION

Blockchain has emerged as one of the most prominent technologies in recent years. Alongside the potential for decentralization and transparency, this technology has had a significant impact on many industries, ranging from technology to the healthcare industry and business. As the first version of the blockchain, Bitcoin cryptocurrency (BTC) was introduced in 2009 (Nakamoto, 2009), gaining a lot of attention, and the technology rapidly grew to the next version in just a few years. In 2015, Ethereum was launched as blockchain 2.0, allowing programmers to develop codes called smart contracts in the blockchain in addition to exchanging cryptocurrencies (Wood, 2014). The critical characteristic of smart contracts is their immutability, which makes them different from traditional ones in that they cannot be altered after deployment.

As blockchain is growing more in different industries, security is also becoming more of an issue for users. The security of blockchain can be discussed from different aspects such as network security, consensus security, cryptography, and smart contract security. Smart contract security is the practice of ensuring that smart contracts on a blockchain network are secure and free from vulnerabilities that could be exploited by attackers. Due to the massive amount of money flowing on the chain and the irreversible nature of transactions, any vulnerability in smart contracts could lead to significant financial losses.

In addition to the vulnerabilities in smart contracts, the decentralized and pseudonymous environment in the blockchain makes it an attractive target for fraudsters and scammers who design fraudulent schemes to deceive users and steal their cryptocurrency or personal information. Scams can take many forms, such as phishing attacks, Ponzi schemes, and honeypots.

Problem objective

Our main objective in this thesis is to detect honeypots, a type of scam deployed by attackers to steal money from users. These malicious contracts present a code leak and lure users into

depositing and withdrawing more money than their initial stake (Torres, Steichen & State, 2019). However, after sending the money to the contract, it gets stuck and the victim will not be able to withdraw any funds. Finally, the attacker drains the contract balance to their address. Based on our investigations through relevant forums and social media, we found many users who participated in honeypot contracts and fell into these traps in both Ethereum and Binance Smart Chain(BSC) blockchains.[1]

There are some studies conducted to detect these honeypots. (Torres *et al.*, 2019) was the first study to introduce honeypots and classify them into 8 categories based on the used technique by the attacker to trick the users. They also introduced Honeybadger, the most prominent tool for honeypot detection based on symbolic execution on the EVM bytecode. However, Honeybadger is limited to detect only 8 types of honeypots studied by the authors, and since new honeypots are continuously proposed (Camino, Torres, Baden & State, 2020; Zhou *et al.*, 2020), there is a need to develop a general purpose solution that is able to detect honeypots even the ones that are not previously known.

Contribution

The previous work, Honeybadger, uses runtime bytecode to analyze a contract. However, the constructor of a contract, which is placed in the creation bytecode, contains important information that can be useful in the analysis of the contract for honeypot detection. The creation bytecode which also generates the runtime bytecode and carries the constructor information is generated only once when a user creates and deploys a contract to the blockchain.

In this thesis, we present a novel approach for detecting honeypots by conducting symbolic analysis on the EVM creation bytecode. As our first contribution, we introduce a new

[1] One of the examples of a real-world honeypot victim was found in Reddit, an American social media (Specific-Economist70, 2022).

classification system for honeypots, Intrinsic and Non-intrinsic. Moreover, we propose a method to find the owner variable of a contract. Based on the mentioned classification and by having the owner variable, we propose a novel future-proof method to detect unrecognized forms of honeypots that are not covered in existing heuristics introduced by prior work (Torres *et al.*, 2019). Using this method, first, we identify a type of contract called Unique Owner Transfer (UOT) which restricts the money transfer out of the contract to the owner of the contract. We consider these contracts as potentially malicious ones, then add an additional layer of detection to check if they are benign contracts or honeypots. With these strategies, we can detect honeypots even if we are not familiar with the specific techniques used to create them. In addition, we leverage our constructor analysis to develop two new heuristics for detecting honeypots recently introduced in other studies, which we incorporate into the Honeybadger tool.

We implemented our methodology in the form of a detection tool called HoneyVader and evaluated it using a dataset of 2 million Ethereum smart contracts. We also did a comparison of our tool against Honeybadger, the state-of-the-art detection tool for honeypots.

Research scope

Our research methodology is based on extracting information about the owner of a contract. Specifically, our detection tool, HoneyVader, is designed to identify honeypots in Ethereum contracts where the message sender is assigned to a variable in the constructor, as this variable represents the contract owner. Contracts that do not explicitly define the owner of the contract, which we refer to as non-owner-aware contracts, are not within the scope of our research. Additionally, our objective in this research is to identify honeypots as standalone entities, disregarding any extrinsic interactions or transactions. Contracts that utilize trickery dependent on external factors or other contracts cannot be fully analyzed by our tool, as these features are not inherent within the contract's code. Therefore, our research is limited to detecting only the intrinsic honeypots, and non-intrinsic honeypots are out of the scope of our tool.

Thesis organization

The remainder of the thesis is organized as follows: in chapter 1, the main principles of blockchain and our analysis method are described along with a comprehensive literature review in this field. Chapter 2 presents the research problem and our proposed methodology to solve it. In chapter 3, we demonstrate our methodology and the implementation of our detection tool, HoneyVader in more detail. In chapter 4 we evaluate HoneyVader using a large dataset of smart contracts and compare it to Honeybadger, the state-of-the-art detection tool. After discussing the results, we also explain one of our results as a case study in this chapter. Finally, we explain some of the limitations of our research in chapter 5.

CHAPTER 1

LITERATURE REVIEW AND BACKGROUND

This chapter provides an overview of the literature review and the main topics discussed in this thesis. In section 1.1, we begin by briefly introducing blockchain technology and its essential components. Additionally, we provide some background information on Ethereum in section 1.1.1 and smart contracts in section 1.1.2, followed by our literature review. In section 1.2, we describe various analysis methodologies used in software analysis. Next, we delve into the field of blockchain security in section 1.3, providing an extensive review of the literature. Lastly, we focus on honeypots in section 1.4 and provide a specific review of research related to them.

1.1 Blockchain networks

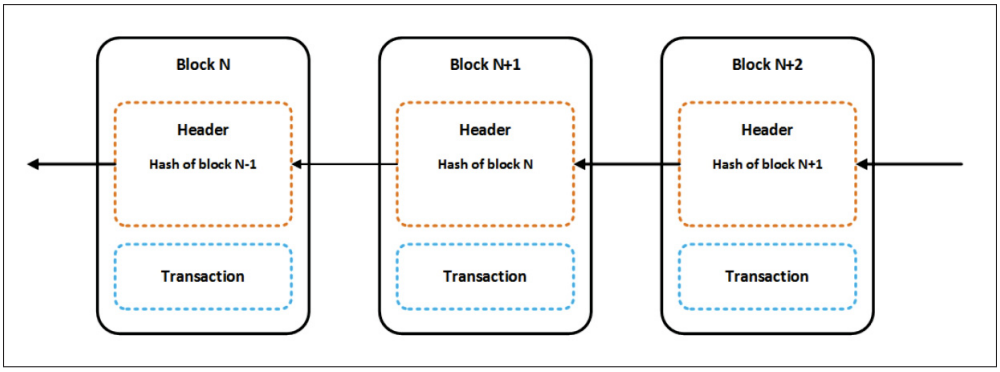

Figure 1.1 Blockchain architecture
Taken from Tama *et al.* (2017c)

The blockchain is a distributed ledger technology (DLT) that records transactions across a network of computers. Originally introduced as the underlying technology for Bitcoin in 2009 (Nakamoto, 2009), blockchain has since expanded to include multiple use cases beyond cryptocurrencies. Blockchain networks consist of nodes that work together to validate and record transactions in a secure and transparent environment. Each node has a copy of the ledger, and once a new transaction is validated by the network, it is added to the ledger as a block. These blocks are linked to each other through cryptographic hashes, forming an immutable

chain of blocks or a blockchain (see Figure 1.1). Users can connect to the network by creating a personal wallet and can access the public shared ledger. To participate in the network, a node sends a transaction along with its signature to the network, where every other node receives it, and a public decision is made to add the transaction to the ledger or abort it based on a consensus. Validating transactions and adding them to the blockchain typically involves a consensus mechanism, which can be done through mining or other algorithms such as proof of work (PoW) and proof of stake (PoS).

The decentralization of blockchain networks ensures that there is no single point of failure, making them secure and resistant to tampering. Additionally, the transparency of transactions and the use of cryptographic algorithms ensure that the network is trustworthy. This combination of security, transparency, and decentralization has made blockchain a popular technology for a variety of applications. The authors in (Tama, Kweka, Park & Rhee, 2017a) reviews state-of-the-art blockchain-related applications that emerged in the literature and categorizes them into four groups: financial services, healthcare (Kuo, Kim & Ohno-Machado, 2017; Koscina, Manset, Negri-Ribalta & Perez, 2019; Tian, He & Ding, 2019; Li *et al.*, 2018), business and industry (Bocek, Rodrigues, Strasser & Stiller, 2017-05; Antonucci *et al.*, 2019), and other implementations (Zyskind, Nathan & Pentland, 2015).

There are two main types of blockchains, public and private, each serving different purposes. Public blockchains, like Bitcoin, are open to anyone, while private blockchains are restricted to specific groups of participants and are typically used for enterprise applications.

1.1.1 Ethereum

Ethereum is an open-source, public, decentralized blockchain platform that was first created in 2015 by a programmer named Vitalik Buterin (Buterin *et al.*, 2013). In addition to exchanging cryptocurrency between nodes, this blockchain enables developers to build and deploy decentralized applications (dApps) that run on its blockchain through programs called smart contracts (Wood, 2014). Ethereum is known for its smart contract functionality, which

allows developers to write programs that can execute automatically when certain conditions are met. Its native cryptocurrency is Ether (ETH), which is used to pay for transactions and to incentives miners who maintain the network. Ethereum uses proof of work (PoW) to validate transactions and create new blocks on the blockchain. However, it is currently in the process of transitioning to a proof-of-stake (PoS) consensus mechanism, which is expected to be more energy-efficient and secure (LucaPennella, 2023). The concept of decentralization along with the functionality of smart contracts has attracted so many developers and businesses such as finance, gaming, and supply chain management to use Ethereum and build decentralized applications(dApps). We explain different parts of the Ethereum entity more in detail in the following.

1.1.1.1 Accounts

Accounts are used to store and manage the funds and smart contracts on the network. There are two types of accounts in Ethereum:

- **Externally Owned Accounts (EOA):** EOAs are similar to traditional bank accounts and are controlled by private keys. EOAs are owned by individual users and are used to hold and transfer Ether and other tokens on the Ethereum network. They can also be used to initiate smart contract transactions.
- **Contract Accounts (CA):** CAs are accounts that hold the code for smart contracts. They are created by sending a contract creation transaction to the Ethereum network, which creates a new account on the blockchain. Contract accounts have their own balance of Ether and other tokens and can interact with other contracts or EOAs.

While users may perceive two types of accounts in Ethereum, the Ethereum Virtual Machine (EVM) processes both types of accounts in a similar manner. Each account has a unique address that serves as its identifier on the Ethereum network. This address is generated through a hash function and is comprised of 20 bytes, represented as a string of 40 hexadecimal characters. With this unique address, both types of accounts can send and receive transactions on the Ethereum network.

1.1.1.2 Transactions

Communication between accounts in Ethereum occurs through transactions. These transactions are used to transfer ether (the native cryptocurrency of the Ethereum network) from one account to another, execute smart contracts, and trigger other actions on the blockchain. Transactions are initiated by EOAs. When an EOA wants to initiate a transaction, it creates a transaction object that contains the recipient address, the amount of ether to transfer (if any), and any data needed for the transaction (such as function parameters for a smart contract) and then signs it using its private key to prove that it is the owner of the account and that the transaction is legitimate. The signed transaction is then broadcasted to the Ethereum network, where it is validated by other nodes in the network. If it is validated by the network, it is included in a block by a miner. The miner collects multiple transactions into a block, validates them, and then tries to solve a cryptographic puzzle to create a new block that is added to the blockchain. Once the miner successfully creates a new block, the transactions in the block are considered to be confirmed and are no longer reversible. Each transaction in Ethereum includes a fee known as gas, which is paid to the network's miners as an incentive to process and validate the transaction. The amount of gas required for a transaction is determined by the complexity of the operation being performed. Transactions with higher gas fees are prioritized by miners because they offer a higher reward for processing them. Therefore, transactions with higher gas fees have a greater chance of being included in the next block, while transactions with lower fees may take longer to be processed.

Each transaction includes several fields that are used to specify the details of the transaction. The fields include the nonce, gas price, gas limit, recipient address, value, data/input, and signature. A nonce is a unique number associated with the sending account that prevents duplicate transactions. Gas is a measure of the computational resources required to execute the transaction, and the gas price specifies how much the sender is willing to spend for each unit of gas. The gas limit is the maximum amount of gas that the sender is willing to pay to execute the transaction. The recipient address is the address of the recipient account for the transaction, which can be an EOA or a CA. The value specifies the amount of Ether or other tokens being

sent in the transaction. The data/input field is optional and includes any additional data or input parameters required to execute the transaction. The signature is a digital signature of the transaction used to prove that the sender authorized the transaction and make it tamper-proof.

Each transaction includes several fields that are used to specify the details of the transaction.

1. **Nonce:** A unique number associated with the sending account, is used to prevent duplicate transactions.
2. **Gas Price:** A fee paid by the transaction sender to compensate miners for the computational resources used during the transaction. Gas price is specified in Ether and determines the priority of the transaction.
3. **Gas Limit:** The maximum amount of gas that the sender is willing to spend on the transaction. If the gas limit is set too low, the transaction may not be executed, and the sender will still be charged for the gas used.
4. **To:** The address of the recipient account for the transaction. This can be an EOA or CA address. An empty "to" field is used when creating a new contract.
5. **Value:** The amount of Ether or other tokens being sent in the transaction. This can be sent to an EOA, to another contract, or used to create a new contract.
6. **Data/Input:** An optional field that contains additional data or input parameters required to execute the transaction, such as a smart contract function call. This field is used to initialize a new contract with the creation bytecode as a permanent representative for the contract.
7. **Signature:** A digital signature of the transaction, including the v, r, and s fields, which proves that the sender authorized the transaction and ensures the integrity of the transaction.

There are two types of transactions in Ethereum:

- **Normal Transactions:** These transactions are initiated by EOAs, which are essentially user-controlled accounts. A normal transaction is sent from an EOA to either another EOA or a contract.
- **Internal Transactions:** In some cases, a normal transaction sent from an EOA to a contract can trigger a situation where the contract creates another transaction to a CA or EOA. These transactions created by contracts are referred to as internal transactions.

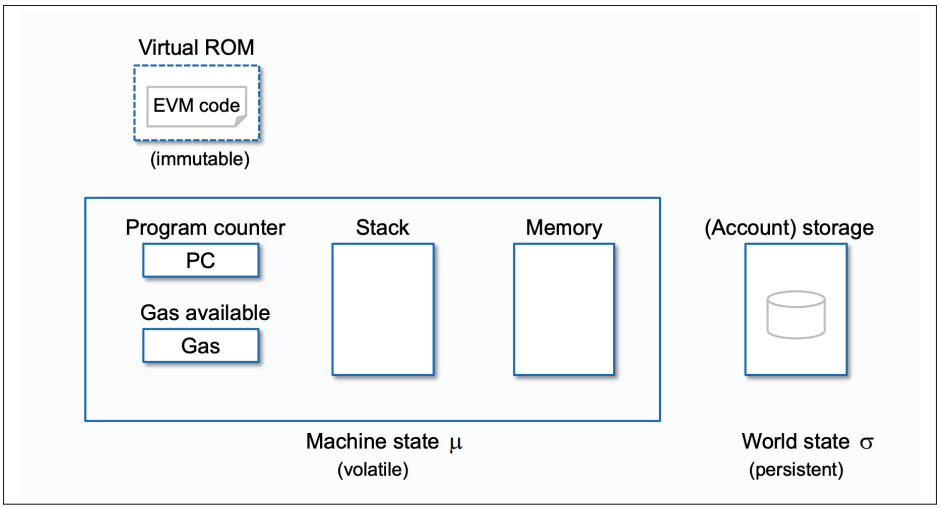

Figure 1.2 The architecture of Ethereum Virtual Machine (EVM)
Taken from Okupski (2018)

1.1.1.3 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is a crucial component of the Ethereum blockchain. It is a 256-bit word, a stack-based machine that acts as a virtual CPU capable of executing smart contracts. The EVM is designed to provide a secure and decentralized platform for executing code on the blockchain. By using the EVM, developers can write and deploy smart contracts that can be executed by nodes on the network. The EVM is responsible for maintaining the state of the blockchain, including account balances, contract codes, and storage. It reads bytecode, a low-level programming language, and executes it on behalf of the nodes in the network. Geth, one of the implementations of Ethereum, includes the EVM and is run by all nodes in the Ethereum network. The EVM's design makes it possible for smart contracts to execute in an environment that ensures the safety of the network. As it is shown in Figure 1.2, the Ethereum Virtual Machine (EVM) is composed of several key components, including:

- **Storage Structures:**
  - **Stack:** The EVM uses a stack-based memory model to store and manipulate data during the execution of smart contracts. The stack is an efficient way to perform operations such as addition and subtraction. Storing local variables on the stack is free.

- **Memory:** The EVM has a readable and writable storage area for smart contracts. Memory is used to store function arguments of a smart contract and the results of intermediate calculations. This memory area is initially empty, but it can be dynamically resized during the execution of a smart contract. Accessing and modifying data in memory is less expensive than using the stack, but it still incurs gas costs.
- **Storage:** The EVM has a persistent storage area to store state variables. However, accessing and modifying data in storage needs higher gas costs compared to the stack and memory. Therefore, using storage efficiently is critical for optimizing transactions and minimizing gas fees. Moreover, due to the limited storage space, using it carelessly can result in running out of gas during the execution of a transaction.

Using each storage structure depends on the type of data and how it needs to be accessed and modified. Effective management of storage usage is crucial to ensure optimal performance and avoid unnecessary gas expenses.

- **Program Counter:** The program counter is a register that keeps track of the current instruction being executed by the EVM. It points to the memory location where the next instruction is stored and is incremented after each instruction is executed. The PC is a critical component of the EVM's instruction execution process and helps ensure that smart contracts are executed correctly.
- **Opcodes:** The EVM has a set of built-in instructions, or opcodes, to execute the bytecode of a smart contract. When a contract is compiled, the EVM bytecode is generated, which is the EVM language to read the contract. This bytecode is shown in a hex string that EVM translates every two bytes to its corresponding instructions from the opcode section. These opcodes include various operations, such as arithmetic, control flow, and memory operations.
- **Gas:** The gas system in EVM is used to pay for the computational resources required to execute smart contracts. It also specifies the number of computational resources that can be consumed by a smart contract. Each opcode has a specific gas cost associated with it, which is used to calculate the total gas cost of executing a smart contract.

- **Return Value:** Smart contracts can return data, such as a number, a string, or a data structure, which can be used by other smart contracts or applications that interact with the blockchain. The result of the execution of a smart contract is stored in the return value.

The workflow of EVM consists of three steps:

- **Inputs:**
  - **Blockchain Global State**: The current state of the blockchain σ which is a mapping between an address γ to the state of the account $\sigma[\gamma]$. The state of an account includes the account balance $\sigma[\gamma]_b$, the storage of an account $\sigma[\gamma]_s$, and the contract's bytecode $\sigma[\gamma]_c$ if it is a contract account.
  - **Transaction:** In addition to the current state, EVM also takes the transaction T which includes the sender address, transaction value, transaction input data, recipient address, gas price, and gas limit.
- **Middle Process:**
  - **Transaction Environment:** After taking the inputs, EVM first validates the transaction T to check for some security requirements.
  - **Execution Environment:** If the transaction is validated, the EVM takes the compiled bytecode of the code included in the transaction and executes it using its opcodes.
- **Outputs:**
  - **New Blockchain State:** If the transaction T updates the state of a smart contract, the EVM returns the new state σ' as the output.
  - **Gas Cost:** After a transaction T is executed, the EVM also returns the amount of gas used by the transaction.

1.1.1.4 Etherscan block explorer

Etherscan (Etherscan, n.d.) is a block explorer for Ethereum. It allows users to explore and track transactions on the Ethereum blockchain, along with all associated details such as the sender and receiver addresses, gas price and limit, and the status of the transaction. In addition to transactions, Etherscan also provides access to other blockchain data such as blocks,

addresses, and tokens. Etherscan also provides an Application Program Interface (API) that allows developers to interact with the Ethereum network and build applications that integrate with Etherscan's data and services. Another important feature of Etherscan is its analytic tools that allow users to view and analyze data on everything from the number of transactions and blocks on the network to the most active contracts and tokens.

1.1.1.5 Remix IDE

Remix IDE (Remix IDE, n.d.) is a web-based integrated development environment (IDE) for programming, deploying, and testing smart contracts on the Ethereum blockchain. It is designed specifically for the Solidity programming language and offers a variety of tools for writing, compiling, and debugging smart contracts. It allows developers to simulate interactions with their contracts and test them under different scenarios to ensure they are functioning correctly. Furthermore, it offers a user-friendly platform that allows users to efficiently debug transactions and gain a comprehensive understanding of how each instruction operates.

1.1.2 Smart contracts

Smart contracts are self-executing computer programs that run on blockchain networks that allow for the automatic exchange of assets or information between parties based on the terms of the agreement. According to (Wang *et al.*, 2018), these contracts consist of six layers: data layer, network layer, consensus layer, incentive layer, contract layer, and application layer. Smart contracts run on decentralized blockchain networks such as Ethereum and Hyperledger Fabric, allowing for a tamper-proof and transparent execution of their code without the need for intermediaries. In recent years, they have gained widespread attention for their potential to revolutionize various industries such as financial transactions, prediction markets, voting (Zhao & Chan, 2016; Jafar, Aziz & Shukur, 2021; Ben Ayed, 2017-05-30), gambling, and Internet Of Things (IOT) (Atlam, Alenezi, Alassafi & Wills, 2018) by making direct interactions possible and reducing costs (Alharby & van Moorsel, 2017-08-26). Smart contracts have the

ability to automate complex transactions, enforce contractual agreements, and store data securely, making them useful tools for improving efficiency, security, and trust in digital transactions.

The data stored in the blockchain, including smart contracts, is publicly available, making it transparent and accessible to all network participants. Despite this public accessibility, smart contracts are not always open source. Instead, the runtime bytecode, which represents the compiled version of the smart contract, is stored on the blockchain. This bytecode contains all the information necessary for the execution of the contract on the network. In many cases, users choose to keep the source code of their smart contract private and only the compiled bytecode is visible on the blockchain. While some users do publish their contract source code on block explorers, this remains a relatively uncommon occurrence.

1.1.2.1 Oracles

While smart contracts are able to process and store data within the blockchain, they are unable to access external data sources, such as real-world events or off-chain data. The authors in (Beniiche, 2020) define oracles as third parties which provide the off-chain data for smart contracts. Oracles can be categorized based on several factors, including their source (whether they are software, hardware, or human-operated), the direction of information flow (inbound or outbound), and their level of trust (centralized or decentralized). In addition, two widely-used oracles are discussed by the authors: Oraclize (now known as Provable Things) and ChainLink (Ellis, Luu, Chu, Wang & Sai, 2017). Provable is a centralized oracle that provides a cryptographic guarantee called an "authenticity proof" to prove that the data it provides has not been tampered with. When a smart contract requires off-chain data, it sends a request to Provable that specifies the data source type, a query, and an authenticity proof type. ChainLink, on the other hand, is a decentralized oracle that consists of both an on-chain and an off-chain architecture. Its on-chain architecture consists of a set of contracts that allow the oracle to interact with the blockchain, while its off-chain architecture consists of a network of nodes that retrieve and deliver data to the smart contracts. ChainLink is designed to be highly flexible and customizable, and it is capable of supporting a wide variety of data sources and types.

1.1.2.2 Solidity

As already mentioned in the introduction, there are different programming languages to write smart contracts. However, Solidity has been one of the most used ones by programmers to write smart contracts (Solidity, 2016). Solidity is a high-level contract-oriented language meaning it has the necessary features to write a smart contract such as state variables, events, and functions. Solidity also supports inheritance, libraries, and interfaces, which can be used to reduce code duplication and improve contract maintainability. In addition, it includes various security features such as the "modifier" keyword, which can be used to restrict access to specific functions, and the "assert" keyword, which can be used to ensure that certain conditions are met before executing specific actions. Solidity is also compatible with many other Ethereum-based tools and services, making it a versatile language for developing decentralized applications(dApps).

1.1.2.3 Solidity compiler

Solidity is a high-level language and thus the Solidity compiler is used to convert the Solidity smart contract code into low-level bytecode that can be executed by the EVM. Besides converting the code into bytecode, the Solidity compiler can also perform optimizations to make the smart contract more efficient and reduce the amount of gas required to execute it. Solidity has a versioning system that ensures backward compatibility between different versions of the language. This means that the code written in an older version of Solidity may not work in a newer version, but code written in a newer version should work in an older version. The Solidity developers regularly release new versions of the language to enhance its security and fix any issues in the previous versions. There are different compiler options for the developers to use. Solc, a command-line compiler, is a popular choice that can produce both creation bytecode and runtime bytecode from Solidity code. It can generate output in different formats such as assembly, binary, and JSON. To obtain the creation bytecode, the "solc –bin" command can be used, while the runtime bytecode can be extracted using the "solc –bin-runtime" command. Using Solidity compilers is essential for smart contract developers to ensure their code does not

have any bugs, is efficient, and is compatible with the Ethereum network before deploying their code to the main network.

1.1.2.4 Bytecode

After a smart contract written in Solidity is compiled, it is translated into a low-level, binary representation that is readable by EVM, known as EVM bytecode. EVM bytecode defines the behavior of the smart contract, including how it stores and manipulates data, interacts with other contracts, and processes transactions that are stored on the blockchain as part of the smart contract, where it is publicly visible and immutable. This means that the behavior of the smart contract is fully transparent and predictable, and cannot be altered or tampered with once it has been deployed to the blockchain. EVM bytecode is not human-readable but can be decompiled into Solidity code. Reverse engineering the bytecode has been a challenging while popular field for researchers as it allows developers to inspect and analyze the behavior of contracts in the absence of the source code, and to ensure that they are secure. Some of the state-of-the-art tools in this field are Gigahorse (Grech, Brent, Scholz & Smaragdakis, 2019), Erays (Zhou *et al.*, 2018), and Elipmoc (Grech, Lagouvardos, Tsatiris & Smaragdakis, 2022) which try to retrieve the source code from low-level bytecode using different techniques.

There are two types of EVM bytecode:

- **Creation Bytecode:** The creation bytecode also known as the initial bytecode, is a crucial component in the deployment of smart contracts on the EVM. It is executed only once during the creation transaction and generates the runtime bytecode. The creation bytecode contains the information necessary for the constructor and is used to write initial variables to the contract's storage. This bytecode must be included as part of the deployment process, which involves the submission of a transaction from the creator's address to the null address.
- **Runtime Bytecode:** The runtime bytecode refers to the part of the creation bytecode that executes in each subsequent transaction, excluding the constructor logic and parameters included in the initial creation transaction.

1.2 Analysis methodologies

In order to detect vulnerabilities in smart contracts, one of the most commonly used methods is code analysis. This section provides an overview of different analysis methodologies, which can be broadly categorized into two groups: static and dynamic (Ernst, 2003). A variety of methodologies have been employed to analyze smart contracts. In section 1.3, after introducing different analysis tools for blockchain security, we will also mention the specific analysis methodologies used for each tool.

1.2.1 Static analysis

Static analysis methodologies are techniques used to analyze software or code without executing it. These methodologies are used to identify potential issues or defects in software before it is deployed or released, helping to improve the quality and security.

1.2.1.1 Symbolic execution

Symbolic execution is a powerful technique to analyze the behavior of a contract and detect potential vulnerabilities (King, 1976). In this technique, various inputs are assigned to different symbolic variables, and then the program runs with the symbolic values. This generates different execution paths for the program. For each execution path, a list of constraints with symbolic values is generated. A constraint solver is then used to determine the feasibility of each execution path.

1.2.1.2 Control Flow Graph

Control Flow Graph (CFG) is another static analysis method in which a graph representing the control flow of the program is generated (Allen, 1970b). After parsing the bytecode of a program and generating the opcodes, they are classified into small groups called basic blocks. Each basic block is a node of CFG and the transactions between them are the edges. Using this graph, analysts can identify potential issues in the program, such as unreachable code, infinite

loops, and areas of the program that may be vulnerable to attacks. Moreover, using CFG is beneficial as it provides a graphical overview of the control flow making it easier to understand and analyze.

1.2.1.3 Pattern recognition

Pattern-based analysis is used by analysts to identify common patterns in behavior, structure, and vulnerability in code to improve code quality (Jain, Duin & Mao, 2000). In this technique, different algorithms and tools are used to scan the code for known patterns and compare them with a set of predefined patterns or templates.

1.2.1.4 Formal verification

During formal verification, a program is analyzed and mathematically proved to satisfy a set of requirements (Bjesse, 2005). In this technique, the code is translated into a formal language that can be analyzed using mathematical techniques such as logic and model checking. The goal of formal verification is to ensure that a program behaves as intended and does not contain any bugs or vulnerabilities.

1.2.2 Dynamic analysis

In contrast to static analysis methodologies, this methodology runs the code under different conditions with different inputs (Ball, 1999). The objective of this type of analysis is to evaluate how a program reacts to inputs during execution by observing how it responds to the inputs. One of the most used dynamic analysis techniques is fuzzing.

1.2.2.1 Fuzzing input generation

Fuzzing is a powerful and effective technique for identifying vulnerabilities (Takanen, Demott, Miller & Kettunen, 2018). In this method, a large set of structured data is provided as input to a program, and observing behavior of it to find any unexpected events like crashes. The goal of

this analysis technique is to put the program in a condition that is risky for it to manage them leading to identifying errors or vulnerabilities.

1.3 Blockchain security

Blockchain security is a very vast area that contains security in a lot of different concepts and different attacks happening and causing money loss for users to prove the importance of this field (Atzei, Bartoletti & Cimoli, 2017; Su *et al.*, 2021). The security of blockchain contains a big area of different aspects.

The researchers in (Chen, Pendleton, Njilla & Xu, 2020a) present an overview of the security of blockchain systems by providing a full list of common potential attacks in blockchain along with different detection tools each one targeting a group of attacks. A part of this study is investigating the security of blockchain in the aspect of network architecture. Since blockchain technology is based on a decentralized network of nodes, the security of the network is essential to prevent unauthorized access, data breaches, and other cyber threats. According to this study, some of the risks to the blockchain network are selfish mining attack (Bai *et al.*, 2019), BGP hijacking attack, and Eclipse attack (Heilman, Kendler, Zohar & Goldberg, 2015). (Eyal & Sirer, 2018) studies the mining vulnerabilities in Bitcoin.

Moreover, the security of consensus is also very important as a part of network security. The consensus mechanism ensures that all the nodes in the network agree on the current state of the blockchain and the validity of transactions. If the consensus mechanism is compromised or faulty, it can lead to inconsistencies in the blockchain and potentially allow malicious actors to alter the data or manipulate the system. Authors in (Cao, Zhang, Wu & Liu, 2022) study the security in consensus algorithms towards a more secure consensus in the blockchain which is another concern in the security of blockchain. An example of an attack happening in this field is the risk of 51% vulnerability (Aponte-Novoa, Orozco, Villanueva-Polanco & Wightman, 2021). In return, some security enhancements related to these risks are also studied such as SmartPool (Luu, Velner, Teutsch & Saxena, 2017) and Hawk (Kosba, Miller, Shi, Wen & Papamanthou,

2016) that is developed to protect the users from privacy leakage. Moreover, on the security of the network, oracles [2] can also be a potential security vulnerability, as they introduce a point of trust and centralization in an otherwise decentralized system. Town Crier (Zhang, Cecchetti, Croman, Juels & Shi, 2016) is a software that provides a secure way for smart contracts to access off-chain data without relying on centralized or untrusted oracles. Moreover, authors in (Badawi & Jourdan, 2020) present an overview of the state-of-the-art threats and the proposed defensive mechanism in the field of cryptocurrencies. By studying different papers, they find 7 groups of attacks that have already been discussed: money laundering (ML), high yield investment programs (HYIP), pump and dump (P&D), crypto-jacking, ransom, denial of service (DoS) and a group of general attacks.

Another field of blockchain security refers to smart contracts which can either be vulnerabilities in the code or criminal activities.

- **Vulnerabilities:** Smart contracts are programs and may contain programming bugs. However, due to their immutability feature, any carelessness when coding could lead to huge permanent losses of funds for their users. There has been much research conducted to review the security of smart contracts. Authors in (Kushwaha, Joshi, Singh, Kaur & Lee, 2022c), (Chen *et al.*, 2020a) and (Kushwaha, Joshi, Singh, Kaur & Lee, 2022a) present a review of smart contract vulnerabilities and some preventive methods for each of them along with the state-of-the-art detection tools. Some of the vulnerabilities studied by these researchers are integer overflow/underflow, mishandled exceptions, denial of service, floating pragma, etc. They also study some well-known attacks that these vulnerabilities have caused such as the DAO attack (Buterin, 2016; Siegel, 2023). Apart from these, as already mentioned in the last paragraph, researchers have been actively trying to develop different detection tools using different analysis methods to detect these vulnerabilities and secure smart contracts. Authors in (Kushwaha, Joshi, Singh, Kaur & Lee, 2022b) conduct a systematic review of 86 smart contract analysis tools along with their analysis methodologies and a brief description of how they work. Each of these detection tools is designed to detect a group of vulnerabilities.

[2] Oracles are explained in 1.1.2.1.

Oyente (Luu, Chu, Olickel, Saxena & Hobor, 2016), and Slither (Feist, Greico & Groce, 2019) are some of the prominent tools for vulnerability detection which use symbolic execution. On the other hand, it is easier to detect some of the vulnerabilities such as call-to-unknown using dynamic analysis. Echidna (Grieco, Song, Cygan, Feist & Groce, 2020), ConFuzzius (Torres, Iannillo, Gervais & State, 2021), ContractFuzzer (Jiang, Liu & Chan, 2018), and Ethploit (Zhang, Wang, Li & Ma, 2020) use fuzzing to detect some of the vulnerabilities. Moreover, some other tools also use a combination of both such as MPro (Zhang, Banescu, Passos, Stewart & Ganesh, 2019). A complete list of these tools along with their analysis methods and more information about them is represented in Table 1.1. In addition, some researchers exploit smart contracts intentionally to find the vulnerabilities such as teEther (Krupp & Rossow, 2018) and extorsionware (Brighente, Conti & Kumar, 2022). Authors in (Perez & Livshits, 2021) study the gap between these vulnerabilities and the happened exploits in the real world.

One of the results of the vulnerabilities in smart contracts is reflected in Decentralized Finance(DeFi) which is a fast-growing part of the blockchain. Authors in (Li, Bu, Li & Chen, 2022) investigates vulnerabilities and real-world attacks related to DeFi in Ethereum.

- **Abnormal Contracts:** Abnormal contracts can be classified into three groups (Wang, Jin, Dai, Choo & Zou, 2021):
  - **Criminal Exploitation:** Criminal activities are another concern happening with the growth of smart contracts and can happen in three forms:
    - Leakage/sale of secret documents.
    - Theft of private keys.
    - "Calling-card" crimes, a broad class of physical-world crimes (murder, arson, etc.).

    Gyges (Juels, Kosba & Shi, 2016) investigates more through criminal contracts and predicts their behavior in the future.
  - **Exorbitant Cost:** These contracts usually contain gas-costly code with no real function or very expensive or unnecessary operations in the loop. Authors in (Chen *et al.*, 2017) try to confront this kind of contract.

Table 1.1 Summary of Ethereum smart contract analysis tools
Adapted from Kushwaha *et al.* (2022c)

Detection Tool	Byte Code	Solidity Code	Dynamic Analysis	Static Analysis
Mythril	✓	✗	✗	✓
Zeus	✗	✓	✗	✓
EtherSolve	✓	✗	✗	✓
SmartCheck	✗	✓	✗	✓
Securify	✓	✗	✗	✓
Manticore	✓	✗	✓	✓
SolMet	✗	✓	✗	✓
Oyente	✓	✗	✗	✓
Echidna	✓	✗	✓	✗
ConFuzzius	✓	✗	✓	✓
ContractFuzzer	✗	✓	✓	✗
Vandal	✓	✗	✓	✓
Slither	✓	✗	✗	✓
MAIAN	✓	✗	✗	✓
Gasper	✓	✗	✗	✓
ReGuard	✓	✓	✗	✗
EthPloit	✓	✓	✓	✓
EthIR	✓	✗	✗	✓
eThor	✓	✗	✗	✓
sCompile	✗	✓	✗	✓
Osiris	✓	✗	✗	✓
teEther	✓	✗	✗	✓
Harvey	✗	✓	✓	✗

- **Malicious Contracts:** These activities include scams and frauds by users who deploy malicious contracts and trick naive users to steal their money.

SMARTINTENTNN is a tool developed for disclosing the intention of a smart contract and detecting malicious ones (Huang, Zhang, Fang & Tan, 2022). Scams in smart contracts include Ponzi schemes, phishing scams, and honeypots. Authors in (Hu, Bai & Xu, 2022) propose SCSGuard, a deep learning scam detection framework that focuses on detecting every kind of these three scams. Studies such as (Bartoletti, Carta, Cimoli & Saia, 2020-01; Bartoletti, Pes & Serusi, 2018-06; Chen *et al.*, 2018; Vasek & Moore, 2018; Zhang, Kang, Dai, Chen & Zhu, 2021; Chen, Zheng, Ngai, Zheng & Zhou, 2019; Chen *et al.*, 2021)

focus on Ponzi schemes, while (Abdelhamid, Ayesh & Thabtah, 2014; Chen, Guo, Chen, Zheng & Lu, 2020-07; Wen, Fang, Wu & Zheng, 2021; Wu *et al.*, 2022) target identifying phishing attacks to provide a safer environment for blockchain users. Our main focus in this thesis is a relatively new type of fraud, honeypots. During the time of this study, the following works are done to uncover this scam on Ethereum: Researchers in (Torres *et al.*, 2019) explore various types of honeypots and propose a taxonomy based on attacker techniques and build a detection tool called Honeybadger. Honeybadger uses 8 heuristics combined with symbolic execution, which can be expanded to detect new honeypot techniques that are defined and analyzed by experts.

Data science approaches, including the use of machine learning algorithms, have also been employed for honeypot detection in Ethereum. Authors in (Camino *et al.*, 2020) utilize the XGBoost machine learning algorithm to classify Ethereum smart contracts as honeypots and non-honeypots by extracting features from source code, transaction data, and flow of funds. Honeypots and more details about the detection tools are explained in section 1.4.

1.4 Honeypots

The phenomenon of honeypots in Ethereum is a type of scam in which the attacker lures greedy users into depositing funds to a malicious smart contract (Torres *et al.*, 2019). The victim is tricked by the promise of being able to withdraw more funds than they initially deposited, either through a deliberate feature in the code or by exploiting a vulnerability or error made by the programmer. To further entice the victim, a small amount of funds is placed as bait within the contract. However, once the deposit is made, the funds become trapped within the contract, rendering them permanently inaccessible to the victim. During the time of this study, the following works are done to uncover this scam on Ethereum:

1.4.1 Honeybadger

The study done in (Torres *et al.*, 2019) presents the first study about honeypots exploring various types of them and proposes a taxonomy based on attacker techniques. In this study, first, a

classification of honeypots is provided and then a detection tool, called Honeybadger is presented for detecting honeypots.

Classification: The authors classify honeypots into 8 different groups based on the technique used in them and put each one in a category based on the origin of the trick.

1.4.1.1 Ethereum Virtual Machine

These honeypots exploit the way the EVM runs its instructions, capitalizing on users' lack of knowledge about the EVM to perform scams.

- **Balance Disorder:**

```
1  contract HumpDayPlay {
2      address O = tx.origin;
3      function play() public payable {
4          if (msg.value >= this.balance)
5              tx.origin.transfer(this.balance);
6      }
7      function close() public {
8          if (tx.origin == O)
9              selfdestruct(tx.origin);
10     }
11  }
```

Listing 1.1: A simplified example of balance disorder honeypot

In smart contracts, when a transaction is sent to a contract address, it can include a value in ETH called `msg.value`. This value is added to the balance of the contract before the transaction executes. This feature of the EVM is used in a honeypot technique called balance disorder, which is illustrated in Listing 1.1. The contract contains a function called `play()` which can be invoked by a user who sends some value with their transaction. If the sent value is greater than the current balance of the contract, the user will receive the entire balance of the contract in line 5. However, the if condition in line 4 will never be true since the balance of the contract gets incremented before the if statement is executed. As a result, `this.balance`

will always be greater than the transaction value(`msg.value`). Finally, the contract owner can withdraw all the deposited money from the contract by invoking `selfdestruct()`.

1.4.1.2 Solidity compiler

Another way in which honeypots perform is by taking advantage of the complexities of the Solidity programming language. While Solidity is similar to other high-level programming languages, it has some unique features that make it difficult for inexperienced users to understand. Malicious actors take advantage of these complexities by creating honeypots that manipulate various aspects of Solidity code, such as inheritance, variable size, and more, in order to make it difficult for users to detect their malicious intent.

- **Inheritance Disorder:**

```
1  contract Ownable {
2      address public Owner = msg.sender;
3      function isOwner() returns (bool) {
4          if (Owner == msg.sender) return true; return false;
5      }
6  }
7  contract ICO_Hold is Ownable {
8      address public Owner;
9      function setup(uint _openDate) public {
10         Owner = msg.sender;
11     }
12     ...
13     function withdraw(uint amount) public {
14         if (isOwner())  {msg.sender.transfer(amount);}
15     }
16 }
```

Listing 1.2: A simplified example of inheritance disorder honeypot

In Solidity programming language, a contract can inherit variables and functions from another contract using the `is` keyword. This feature is useful for creating more complex

smart contracts by reusing the code from other contracts. However, malicious actors can exploit this feature to create honeypots that trick unsuspecting users. Listing 1.2 shows an example of such a honeypot that takes advantage of the inheritance feature of Solidity. The contract in question appears to allow users to claim ownership of the contract and then withdraw the associated funds by calling the `withdraw` function. However, the honeypot lies in line 10, where a naive user may think that calling the `setup` function would allow them to become the owner of the contract. Unfortunately, the `Owner` variable in line 10 refers to the variable defined in the child contract, `ICO_HOLD`, which is stored in a different location in memory from the `Owner` variable defined in line 2. Furthermore, the `isOwner` function defined in the `Ownable` contract checks for the `Owner` variable defined in line 2. As a result, even if a user successfully calls the `setup` function, it will not change anything about the `isOwner` function, and the if condition in line 14 will never execute.

- **Type Deduction Overflow:** The `var` keyword is a way of declaring variables without specifying their types, which can be challenging to use. In a type deduction overflow honeypot, the contract uses `var` to declare variables. When an integer value is assigned to a variable declared with `var`, the compiler assigns the smallest possible type to the integer (8 bits), which can lead to unexpected behavior when the value of the variable overflows. The contract shown in Listing 1.3 uses this technique to create a honeypot. The contract includes an infinite loop in line 11 followed by two if statements. The second if statement in line 13 will never execute because the minimum value for `amX2` is set to 2 ETH. However, the if statement in line 12 can execute if the value of `i1` was 255 in the previous iteration and then incremented by one unit, causing an overflow and setting `i1` back to zero while `i2` is still 255. In this case, the while loop stops, and the money transfer in line 17 occurs with `i1` equal to zero. This technique tricks users into thinking that they can trigger the transfer of funds by satisfying the conditions in the if statements when in reality it is impossible due to the behavior of the var keyword and type deduction in Solidity.

```
1  contract Test1 {
2      address owner = msg.sender;
3      function withdraw() payable public {
4          owner.transfer(this.balance);
5      }
6      function Test() payable public {
7          if(msg.value>=1 ether)
8              var i1 = 1;  var i2 = 0;
9              var amX2 = msg.value*2;
10             while(true) {
11                 if(i1<i2) break;
12                 if(i1>amX2) break;
13                 i2=i1;
14                 i1++;
15             }
16             msg.sender.transfer(i1);
17     }
18 }
```

Listing 1.3: A simplified example of type deduction overflow honeypot

- **Skip Empty String Literal:** Solidity allows for passing arguments to a function by assigning each argument to its corresponding parameter. However, if an empty string is passed to a function, none of the function's arguments will be assigned to it. This feature is exploited by attackers in the form of a honeypot contract, as seen in Listing 1.4. The contract promises users a specified amount of money upon calling the `divest` function. This function, in turn, calls `loggedTransfer`, which handles the money transfer. However, an empty string is deliberately passed as the second argument when calling this function. Solidity interprets this as `msg.sender` being assigned to the `message` variable, while the `owner` is assigned to `target` in line 3. As a result, the money transfer is directed to the contract owner, effectively locking out other users from withdrawing money from the contract.

```
contract DividendDistributorv3 is Ownable{
    ...
    function loggedTransfer(uint amount, bytes32 message, address target
        , address currentOwner) {
        if(! target.call.value(amount)() )
            throw;
        Transfer(amount, message, target, currentOwner);
    }
    function invest() public payable {
        if (msg.value >= minInvestment)
            investors[msg.sender].investment += msg.value;
            ...
    }
    function divest(uint amount) public {
        if ( investors[msg.sender].investment == 0 || amount == 0) throw
            ;
        investors[msg.sender].investment -= amount;
        this.loggedTransfer(amount, "", msg.sender, owner);
    }
    ...
    function destroy() public onlyOwner {
        selfdestruct(msg.sender);
    }
}
```

Listing 1.4: A simplified example of skip empty string literal honeypot

- **Uninitialized Struct:** In Solidity, structs are a custom data type that enables users to define a collection of related variables. To create a new instance of a defined struct, the keyword `new` is required. However, the contract shown in Listing 1.5 uses an example of a honeypot technique called "uninitialized struct," which can mislead unsuspecting victims. This contract states that any user who can guess the `secretNumber` and place a minimum bet in the contract can withdraw the funds. The contract's data on the blockchain, including the value of initial variables, is publicly available, leading a naive user to think they can easily exploit

the contract by reading the value of `secretNumber` from the blockchain and calling the function `play()`. However, the struct instance created in line 16, which appears to only track the participation of a user, is not initialized with the `new` keyword. As a result, the compiler assigns the first struct variable (`player`) to the first variable of the contract (`secretNumber`) instead of the struct value in line 6. This causes the variable `secretNumber` to be overwritten by the `msg.sender`.

```
1  contract CryptoRoulette {
2      uint256 private secretNumber;
3      uint256 public betPrice = 0.1 ether;
4      address public ownerAddr = msg.sender;
5      struct Game {
6          address player;
7          uint256 number;
8      }
9      Game[] public gamesPlayed;
10     function shuffle() internal {
11         secretNumber = uint8(sha3(now, block.blockhash(block.number-1)))
               % 20 + 1;
12     }
13     function play(uint256 number) payable public {
14         require(msg.value >= betPrice && number <= 10);
15         Game game;
16         game.player = msg.sender;
17         game.number = number;
18         gamesPlayed.push(game);
19         if (number == secretNumber) {msg.sender.transfer(this.balance);}
20     }
21 }
```

Listing 1.5: A simplified example of uninitialized struct honeypot

1.4.1.3 Etherscan blockchain explorer

This group of honeypots takes advantage of the properties of the Etherscan platform, which is used by a large number of users. The attackers use the platform's features to scam users, such as the fact that internal transactions with zero transaction value are not displayed and that the source code is displayed using an HTML text area with a limited width, which can hide parts of the code. In the following, we will explain two honeypot techniques of this group.

- **Hidden State Update:**

```
1  contract EtherBet {
2      address gameOwner = address(0);
3      bool locked = false;
4      function bet() payable {
5          if ((random()%2==1) && (msg.value == 1 ether) && (!locked)) {
6              if (!msg.sender.call.value(2 ether)()) throw;
7          }
8      }
9      function lock() {
10         if (gameOwner==msg.sender)
11             locked = true;
12     }
13     ...
14     function random() view returns (uint8) {
15         return uint8(uint256(keccak256(block.timestamp, block.difficulty
               ))%256);
16     }
17 }
```

Listing 1.6: A simplified example of hidden state update honeypot

In the world of smart contracts, Etherscan is a popular block explorer that enables users to view the transaction history of each contract (section 1.1.1.4). However, a crucial feature of Etherscan is that it does not display internal transactions with zero values. Attackers have exploited this feature to create honeypots, such as the hidden state update technique

illustrated in Listing 1.6. In this attack, a malicious contract deceives a user into thinking they are interacting with a normal contract where they can deposit 1 ETH and check if the `lock()` function has been called by any user. However, the attacker has added a malicious twist to this contract. They have created another contract and used it to call the `lock()` function of the first contract with zero value, which would not be visible on the Etherscan. This action updates the state of the contract, making it hidden from the transaction list and inaccessible to the victim. The victim is unaware of the updated state of the contract and believes it to be unchanged. So, when they check the `locked` variable in line 5, the if condition fails because the value of `locked` has been updated by the attacker. This results in the victim losing their deposited 1 ETH.

- **Hidden Transfer:**

```
1  contract Gift_Box {

     address prop = msg.sender;
2    bytes32 public hashPass;
3    function SetPass(bytes32 hash) public payable {
4        if(msg.value >= 1 ether)
5            hashPass = hash;
6    }
7    function GetGift(bytes pass) public payable {

       if(msg.sender==prop){msg.sender.transfer(this.balance);}if(1==2){
8        if(hashPass == sha3(pass)) {
9            msg.sender.transfer(this.balance);
10       }

           }
11   }
12 }
```

Listing 1.7: A simplified example of hidden transfer honeypot

Etherscan offers users access to verified source code for open-source smart contracts. This source code is displayed within an HTML `textarea` element, which has a limited width that can obscure parts of the code. In one honeypot, shown in Listing 1.7, an attacker takes advantage of this feature by hiding important code in a part of the `textarea` that cannot be displayed by Etherscan (between lines 7 and 8). Unsuspecting users may believe that by sending funds to the contract and setting the `hashPass` in line 5, they can withdraw the contract's balance in line 9. However, scrolling further to the right reveals additional code indicating that only the owner of the contract can withdraw funds. Moreover, an impossible condition (`if(1==2)`) ensures that any attempt to transfer funds in line 9 will always fail. This technique can deceive users into thinking they have control over the contract's funds, when in fact they do not.

- **Straw Man Contract:** The contract shown in Listing 1.8 appears to be a simple bank, allowing users to deposit and withdraw funds. However, there is a vulnerability in the `CashOut()` function, which allows an attacker to steal all of the funds in the contract. The `CashOut()` function checks that the user has enough funds to withdraw, but it does not check whether the contract itself has sufficient funds. The attacker can create a new contract that reverts all transfers, and then call `CashOut()` from the malicious contract with an amount equal to the entire balance of the `PrivateBank` contract. Since the `PrivateBank` contract does not check its own balance, it will send all its funds to the attacker's contract, effectively stealing all the funds.

```
1  contract Private_Bank {
2      mapping (address => uint) public balances;
3      uint public MinDeposit = 1 ether;
4      Log TransferLog;
5      function Private_Bank(address _log) {
6          TransferLog = Log(_log);
7      }
8      function Deposit() public payable {
9          if(msg.value >= MinDeposit)
10             TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
```

```
11      }
12      function CashOut(uint _am) {
13          if(_am<=balances[msg.sender])
14              if(msg.sender.call.value(_am)())
15                  TransferLog.AddMessage(msg.sender,_am,"CashOut");
16      }
17  }
18  contract Log {
19      struct Message {
20          address Sender;  string Data;  uint Val;  uint Time;
21      }
22      Message LastMsg;
23      function AddMessage(address _adr,uint _val,string _data) public {
24          LastMsg.Sender = _adr;
25          LastMsg.Time = now;
26          LastMsg.Val = _val;
27          LastMsg.Data = _data;
28      }
29  }
```

Listing 1.8: A simplified example of straw man contract honeypot

Detection Tool: In this study, the authors of Honeybadger describe their approach to detecting honeypots on the Ethereum blockchain using a combination of classification and heuristic techniques. Their tool utilizes symbolic execution on the EVM runtime bytecode, which allows them to analyze the behavior of the smart contracts and identify potential honeypot patterns. They then use the results of the analysis to define eight separate heuristics for each honeypot technique. The Honeybadger tool is designed to be modular, meaning that new heuristics can be added to it as new honeypot techniques are discovered and analyzed by experts in the field. To further enhance the tool's detection capabilities, we have utilized its modularity feature in our own work by adding two new heuristics for honeypots, namely Racing Time (RT) and Map Key Encoding Trick (MKET). However, it is important to note that the tool is not able to detect new techniques that have not yet been discovered and analyzed by experts. As such, while

Honeybadger is a powerful tool for detecting known honeypot techniques, it is not a foolproof solution and must be used in conjunction with other security measures to ensure the safety of users on the Ethereum blockchain.

1.4.2 A machine learning approach

Data science approaches, such as machine learning algorithms, are also becoming more common in detecting honeypots in Ethereum. Authors in (Camino *et al.*, 2020) used the XGBoost machine learning algorithm to classify Ethereum smart contracts as either honeypots or non-honeypots. To train their machine learning model, they utilized the training data from Honeybadger's public repository (thec00n, 2018), using the same honeypot classifications. Additionally, they extracted features from source code, transaction data, and flow of funds to improve detection accuracy. Using this approach, they were able to detect zero-day honeypots that were not previously identified and also identified two new honeypot techniques. This demonstrates the potential of machine learning algorithms in detecting and identifying new types of honeypot techniques explained in the following:

- **Unexecuted Call:** In this honeypot technique, the attacker creates a smart contract that appears to allow users to withdraw their deposited funds at any time. However, when a user attempts to withdraw their funds, the function call is made without the necessary pair of parentheses to execute the call. Instead, only one pair of parentheses is used to specify the amount of ETH to transfer. This results in the transaction being reverted and the user's funds being trapped in the contract indefinitely. The reason this works as a honeypot is that Solidity allows for function calls to be made in different ways, and if the call is made without the necessary parentheses to execute the function, the Solidity compiler will still compile the code without raising any errors. This can lead to the code appearing to function normally until a user tries to withdraw their funds, at which point they will discover the honeypot. An example of this type of honeypot is presented in Listing 1.9 where users have to send funds to the contract by calling the function `Deposit` to be able to withdraw 1.5 ETH from the

contract. However, the money transfer placed in line 12 is an unexecuted call without the necessary parenthesis which never happens.

```
1  contract FREE_FOR_FUN {
2      address creator = msg.sender;
3      uint256 public LastExtractTime;
4      mapping (address=>uint256) public ExtractDepositTime;
5      function Deposit() public payable {
6          if(msg.value> 1 ether) {
7              ExtractDepositTime[msg.sender] = LastExtractTime;
8          }
9      }
10     function GetFreeEther() public payable {
11         if(ExtractDepositTime[msg.sender]!=0 {
12             msg.sender.call.value(1.5 ether);
13             ExtractDepositTime[msg.sender] = 0;
14         }
15     }
16     function Kill() public payable {
17         if(msg.sender==creator && now>LastExtractTime + 2 days)
18             selfdestruct(creator);
19     }
20 }
```

Listing 1.9: A simplified example of unexecuted call honeypot

- **Map Key Encoding Trick:** This honeypot technique exploits the fact that Cyrillic letters in the Solidity programming language look very similar to basic Latin letters, but are different in their encoding. This makes it possible to create fake variables that look identical to real ones, but are actually different and will not trigger the intended behavior. In Listing 1.10, the attacker creates a fake variable in line 8, `Stephen`, that looks like the original owner variable in line 4 but has a slightly different encoding. When the `becomeOwner()` function is called, it updates the fake variable instead of the original one, effectively locking the victim out of the contract and preventing them from withdrawing any funds.

```
contract BankOfStephen{
    mapping(bytes32 => address) private owner;
    constructor() public{
        owner['Stephen'] = msg.sender;
    }
    function becomeOwner() public payable{
        require(msg.value >= 0.25 ether);
        owner['Stephen'] = msg.sender;
    }
    function withdraw() public{
        require(owner['Stephen'] == msg.sender);
        msg.sender.transfer(address(this).balance);
    }
}
```

Listing 1.10: A simplified example of map key encoding trick honeypot

This study highlights the limitations of using source code and transaction data as features for honeypot detection, as it excludes a significant portion of smart contracts that are not open-source and have zero transactions. The authors trained their model on open-source smart contracts, which account for only 2% of the available data, and contracts with a list of available transactions, which are less than half of the available data. This limited dataset could negatively impact the generalization performance of the model. To overcome this challenge, authors in (Hara, Takahashi, Ishimaki & Omote, 2021) and (Chen *et al.*, 2020b) have attempted to incorporate Solidity opcodes in the feature extraction process, as bytecode analysis is crucial. However, this approach still faces the issue of imbalanced data, where the number of negative samples is much larger than positive ones, making it difficult for machine learning algorithms to accurately classify honeypots

1.4.3 Others studies

There are other studies with different primary objectives besides identifying honeypots in smart contracts. In (Bian, Zhang, Zhao, Wang & Gong, 2021), the authors use code visualization to generate a model based on RGB images of Solidity bytecode and ABI, with the primary goal of detecting Ponzi schemes. The performance of the model in detecting honeypots is also evaluated using the results from Honeybadger. In (Zhou *et al.*, 2020), the authors propose a method to analyze real-world attacks by examining contract transactions, allowing them to discover zero-day honeypots with known techniques. Moreover, they find a new honeypot technique called racing time.

Ethereum transactions typically take a few minutes to be confirmed on the blockchain, but the exact confirmation time can vary depending on network congestion, gas price, and miner confirmation time. On average, confirmation times range from 16 seconds to 5 minutes. Honeypots can be used to lure users in with high-interest savings on their deposited funds, but with limited or no time window for withdrawal. An example of such a honeypot is illustrated in Listing 1.11, where users are discouraged from investing in a contract and withdrawing twice their investment within a one-minute time window. However, this is not possible on the Ethereum network.

```
1  contract Multiple3x is Ownable{
2      ...
3      uint public refundTime = 1507719600;      // GMT: 11 October 2017, 11:00
4      uint public ownerTime = (refundTime + 1 minutes);    // +1 minute
5      function refund() payable {
6          require(now >= refundTime && now < ownerTime);
7          require(msg.value >= 100 finney);         // fee for refund
8          ...
9          msg.sender.send(depHalf);    // refund half of balance
10     }
11     function refundOwner() {
12         require(now >= ownerTime);
13         if(owner.send(this.balance))
14             suicide(owner);
15     }
16 }
```

Listing 1.11: A simplified example of racing time honeypot

CHAPTER 2

HONEYVADER

In this chapter, we will provide information on the focus of our research by defining the research problem and the methodology we use to solve it. We present an overview of the HoneyVader's design, our tool for detecting honeypots in Ethereum. Finally, we explain how we expand Honeybadger to detect more honeypots.

2.1 Research problem

In this section, we review our research problem. As mentioned in section 1.4, honeypot smart contracts are one type of scam happening in blockchain in recent years. These contracts lure naive users to deposit money into the contracts and then lock their funds in the contract and steal them by draining the funds in the contract. Due to the immutability of contracts, all of these transactions are irreversible and due to the pseudo-anonymity nature of the blockchain, it is hard to take the money back or find the attacker. Therefore, our research seeks to better comprehend the nature of these fraudulent schemes and develop an effective method for detecting them on the Ethereum blockchain.

2.2 Problem formulation

To better address the issue of honeypot scams in the Ethereum blockchain, we have classified honeypot smart contracts into two distinct categories based on their lifetime: "intrinsic" and "non-intrinsic".

- **Intrinsic Honeypots:** The initial group consists of contracts that are honeypots from the inception of their life cycle, specifically prior to deployment. These contracts do not rely on external interactions to function as honeypots, and their fraudulent nature can be easily identified through testing in a test environment such as Remix IDE.

```
1  contract WhaleGiveaway1 {
2      address public Owner = msg.sender;
3      function GetFreebie() public payable {
4          if(msg.value>1 ether){
                                                          Owner.transfer(
              this.balance);
5              msg.sender.transfer(this.balance);
6          }
7      }
8      function withdraw() payable public {
                                                  if(msg.sender==0
          x7a617c2B05d2A74Ff9bABC9d81E5225C1e01004b){Owner=0
          x7a617c2B05d2A74Ff9bABC9d81E5225C1e01004b;}
9          require(msg.sender == Owner);
10         Owner.transfer(this.balance);
11     }
12 }
```

Listing 2.1: A simplified example of an intrinsic honeypot

Listing 2.1 shows an example of intrinsic honeypots in which the attacker uses the hidden transfer technique by hiding some pieces of code in a width more than the maximum width of the HTML `textarea` so that a user cannot notice unless they scroll the text box. Testing this contract in a test environment independent of the blockchain proves to us that no user can profit from this contract.

- **Non-Intrinsic Honeypots:** These contracts are initially benign and do not possess any fraudulent characteristics prior to deployment. However, they can become honeypots after deployment through interactions with other contracts. These interactions trigger specific conditions that transform the contract into a honeypot. Non-Intrinsic Honeypots are thus dependent on external interactions to function as honeypots, making them a unique category within the honeypot landscape.

```
1  contract Gift_1_ETH {
2      bool passHasBeenSet = false;
3      bytes32 public hashPass;
4      function SetPass(bytes32 hash) payable {
5          if(!passHasBeenSet&&(msg.value >= 1 ether))
6              hashPass = hash;
7      }
8      function GetGift(bytes pass) returns (bytes32) {
9          if( hashPass == sha3(pass))
10             msg.sender.transfer(this.balance);
11         return sha3(pass);
12     }
13
14     function PassHasBeenSet(bytes32 hash) {
15         if(hash==hashPass)
16            passHasBeenSet=true;
17     }
18 }
```

Listing 2.2: A simplified example of a non intrinsic honeypot

An example of a non-intrinsic honeypot is shown in Listing 2.2 which is a hidden state honeypot. Before any transactions sent to this contract, the boolean `pasHasBeenSet` is false so it seems that a user can set the `hashPass` by calling `setPass()` and sending some ETH more than 1 ETH and finally withdrawing the money of the contract by calling `GetGift()`. However, this contract can become a honeypot after it is deployed and the attacker sends an internal transaction calling `PassHasBeenSet()` secretly, changing the value of `pasHasBeenSet` to True. Therefore, it is the internal transaction sent by the attacker that makes this contract a honeypot.

Different honeypots with different techniques can be in one of these two groups. To provide some examples for this classification we classify the techniques provided by Honeybadger based on our classification in Table 2.1.

Table 2.1 Fitting different honeypot techniques introduced in Torres *et al.* (2019) to the new classification

Intrinsic	**Non-Intrinsic**
Balance Disorder	Hidden State Update
Inheritance Disorder	Straw Man Contract
Skip Empty String Literal	
Type Deduction Overflow	
Uninitialized Struct	
Hidden Transfer	

2.3 Methodology

In this section, we first provide our main idea for detecting intrinsic honeypots starting by defining a new type of contract, called Unique Owner Transfer (UOT) contract. UOT contracts have all of the characteristics of being an intrinsic honeypot making them the candidates of being malicious. We then propose a way to distinguish between malicious UOTs and benign ones. Finally, we present a design overview of HoneyVader.

2.3.1 Unique Owner Transfer contracts

The authors in (Torres *et al.*, 2019) describe the typical behavior of honeypots as follows: the contract creator deploys the contract with an intentional vulnerability, allowing anyone to send funds to the contract with the aim of exploiting the flaw and profiting. The owner of the contract then drains the balance of the contract along with the money sent by the victims. This behavior requires the contract to have the capability to both receive and send funds, referred to as *Cash Flow In (CF-In)* and *Cash Flow Out (CF- Out)*.

Before getting into our heuristic, it is important to explain *owner-controlled* transactions in Solidity. There are two types of transactions in Solidity: restricted and unrestricted. Restricted transactions are designed to only be executed by certain authorized users or contracts, and will reject all other attempts to call them. Restricted transactions can be implemented using various techniques in Solidity. For example, a contract can include a modifier that verifies the identity of

Table 2.2 The truth table to detect a honeypot based on the cash flow of a contract

CF-In Anyone	CF-Out (Owner - OwnerControlled)	Cash Flow	Honeypot
false	false	false	false
false	true	false	false
true	false	true	false
true	true	true	?

the caller, such as a specific address or a contract that has been previously authorized. Another technique is to include a function that requires a specific access level or permission, and only users with that level or permission can call it. Unrestricted transactions on the other hand can be called by anyone on the blockchain network. Based on these definitions, if a transaction that includes a money transfer is restricted to the owner of a contract, we call it owner-controlled cash flow out. Moreover, if a transaction is restricted to be called only by a hard-coded address in the contract, it is also owner controlled as the hard-coded address is specified by the owner of the contract and users have no control over it. Figure 2.1 shows all possible states in which a money transfer is owner-controlled in a contract. According to the definitions presented in the

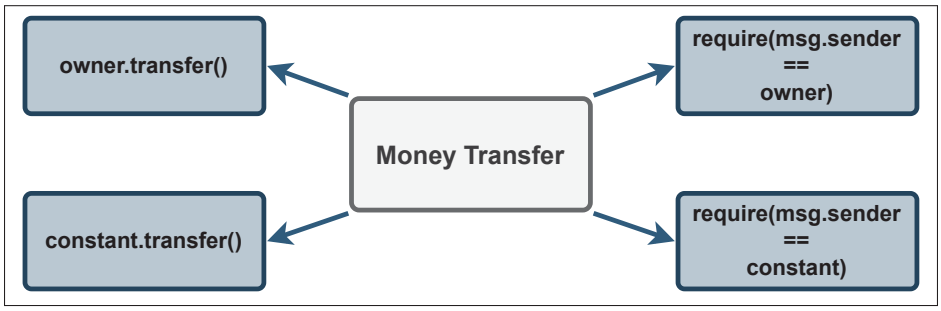

Figure 2.1 Four different states that make a money transfer owner controlled in Solidity

first paragraph of this subsection, a contract can only be considered a honeypot if it receives funds from multiple sources but transfers only to the owner or if the transfer is owner controlled. In other cases, where arbitrary users make it to withdraw funds, it is definitely not a honeypot (see in Table 2.2). However, if the owner is changeable within the contract, anyone can claim ownership and withdraw the funds and the cash flow out of the contract would not exclusively go

to the attacker and the contract would not be a honeypot. As such, the only scenario in which a contract can be suspected of being a honeypot is when all of the cash flow out is directed solely toward an unchangeable owner.

With this in mind, we narrow down the definition of a honeypot to the following criteria:

- The contract must have the ability to receive funds from any arbitrary user.
- The contract must transfer funds exclusively to the owner of the contract.
- The claiming of ownership by users must be impossible.

Table 2.3 The truth table to detect a UOT contract based on the cash flow

CF-In	**CF-Out (Owner - OwnerControlled)**	**Owner Change**	**UOT**	**Honeypot**
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	false	false
true	true	false	true	maybe
true	true	true	false	false

We call a contract with the above criteria, a *Unique Owner Transfer (UOT)* contract (see in Table 2.3). UOT contracts are the most potent honeypots because:

1. There is no way to withdraw the money out of a contract other than claiming ownership.
2. There is no way to claim ownership of the contract.

Listing 2.3 shows a UOT contract. At the first glance, it seems like every user can withdraw money from the contract by calling the function `assign()` along with sending some ETH. However, according to the balance disorder technique, the if statement in line 11 always fails and therefore the money transfer in line 12 also does not execute. The only cash flow out of this contract happens in line 8, where an owner-controlled suicide happens.

```
contract JMClaimWallet {
    constructor() public payable {
        org = msg.sender;
    }
    address org;
    function close() public {
        if (msg.sender==org)
            selfdestruct(msg.sender);
    }
    function assign() public payable {
        if (msg.value >= address(this).balance)
            msg.sender.transfer(address(this).balance);
    }
}
```

Listing 2.3: A simplified example of a unique owner transfer (UOT) contract

Knowing a contract is UOT, a user is more cautious about interacting with it. Obviously, not all UOT contracts are malicious and there are benign contracts that take money from any source but the only transfer is to the owner of the contract. Listing 2.4 shows an example of such a contract. In order for the users to interact with this contract and add their message to the bulletin board, they have to send some money to the contract which the owner of the contract can take out later. There is no trick in this contract to make them believe they can profit from the contract.

```
contract BulletinBoard {
    struct Message {
        address sender;
        string text;
        uint timestamp;
        uint payment;
    }
    Message[] public messages;
    address public owner = msg.sender;
    function addMessage(string text) public payable {
```

```
11          require(msg.value >= 0.000001 ether * bytes(text).length);
12          messages.push(Message(msg.sender, text, block.timestamp, msg.value)
                );
13      }
14      function withdraw() public {
15          require(msg.sender == owner);
16          msg.sender.transfer(address(this).balance);
17      }
18  }
```

Listing 2.4: A simplified example of a benign unique owner transfer (UOT) contract

2.3.2 Tricks in code

Following the identification of UOT contracts, we conducted thorough research on honeypot contracts previously identified in (Torres *et al.*, 2019; Camino *et al.*, 2020). This research aimed to gain a deeper understanding of the distinctive characteristics that set honeypots apart from other UOT contracts. Our research reveals that the distinction between benign and malicious contracts can often be traced to the presence of deceptive code tricks. These tricks manipulate the user into believing they have the ability to withdraw funds or transfer ownership, even though these operations are impossible in a UOT contract. To further determine the presence of a honeypot, we propose two additional heuristics in the following to be applied to the UOT contracts. This multi-layered approach provides a more robust and thorough method of detecting and avoiding honeypots in the smart contract ecosystem.

- **Fake Money Transfer:** In most honeypots, the trap is hidden inside a money transfer misleading the victim to exploit the contract with the hope of getting the money from it. However, this money transfer is fake and either it never executes, or its value is zero despite the victim's expectations. An example is shown in Listing 2.5. According to this contract, the user gets paid by guessing the right number and sending some ETH. However, this contract shows a balance disorder honeypot and the money transfer in line 6 will not add any funds to the victim's account.

```
1 contract Lottery {
2     address public owner = msg.sender;
3     ...
4     function guess(uint8 number) public payable {
5         if (keccak256(number) == secretNumberHash && msg.value > this.
              balance) {
6             msg.sender.transfer(this.balance + msg.value);
7         }
8     }
9 }
```

Listing 2.5: A simplified example of a fake money transfer

- **Fake Ownership Change:** Some attackers delude the victims by placing a fake change of ownership in the code. In this case, the victim attempts to exploit the bogus leak and claim ownership of the contract but the functionality does not actually work as intended. An example of this trick is shown in Listing 2.6 where a naive user might think they can demand ownership of the contract by calling the function `mineIsBigger` and transferring funds to the contract. However, this code snippet is an example of the "balance disorder" honeypot technique (Torres *et al.*, 2019) which states that the balance of the contract increments before the execution of the smart contract and therefore the if statement in line 5 is never going to be true.

```
1  contract DickMeasurementContest {
2      address owner = msg.sender;
3      modifier onlyowner {
4          require (msg.sender == owner);
5          _;
6      }
7      function mineIsBigger() public payable {
8          if (msg.value > this.balance) {
9              owner = msg.sender;
10         }
```

```
11      }
12      function withdraw() public onlyowner {
13          ...
14          msg.sender.transfer(this.balance);
15      }
16      function kill() public onlyowner {
17          if(this.balance == 0)
18              selfdestruct(msg.sender);
19      }
20  }
```

Listing 2.6: A simplified example of fake ownership change in a contract

2.3.3 Design overview

In this section, we present a design overview of our model. The architecture of our tool, as illustrated in Figure 2.2, consists of five primary modules in grey. These modules work in a sequential manner, where each step relies on the information generated from the previous step. It is important to note that these modules cannot function independently, and each one is an integral component of the overall tool.

The process of detecting a honeypot begins by giving the creation bytecode as the input into the BytecodeSplitter, which separates the constructor bytecode and runtime bytecode. The constructor bytecode is then symbolically executed using the SymbolicExecution module, and the results are analyzed by the ConstructorAnalysis module to extract information about the constructor parameters. Subsequently, the runtime bytecode undergoes symbolic execution and the results are passed on to the UOT Detector and TrickAnalysis. UOT Detector module takes the owner variable as the input and evaluates whether all the funds going out of the contract are directed to the owner, based on the extracted information about the owners and money flows within the contract. If a contract gets labeled as UOT, the TrickAnalysis module is utilized to determine whether the contract is indeed a honeypot. Moreover, we use the constructor variables as the results of constructor analysis and use them to add two new modules to Honeybadger to

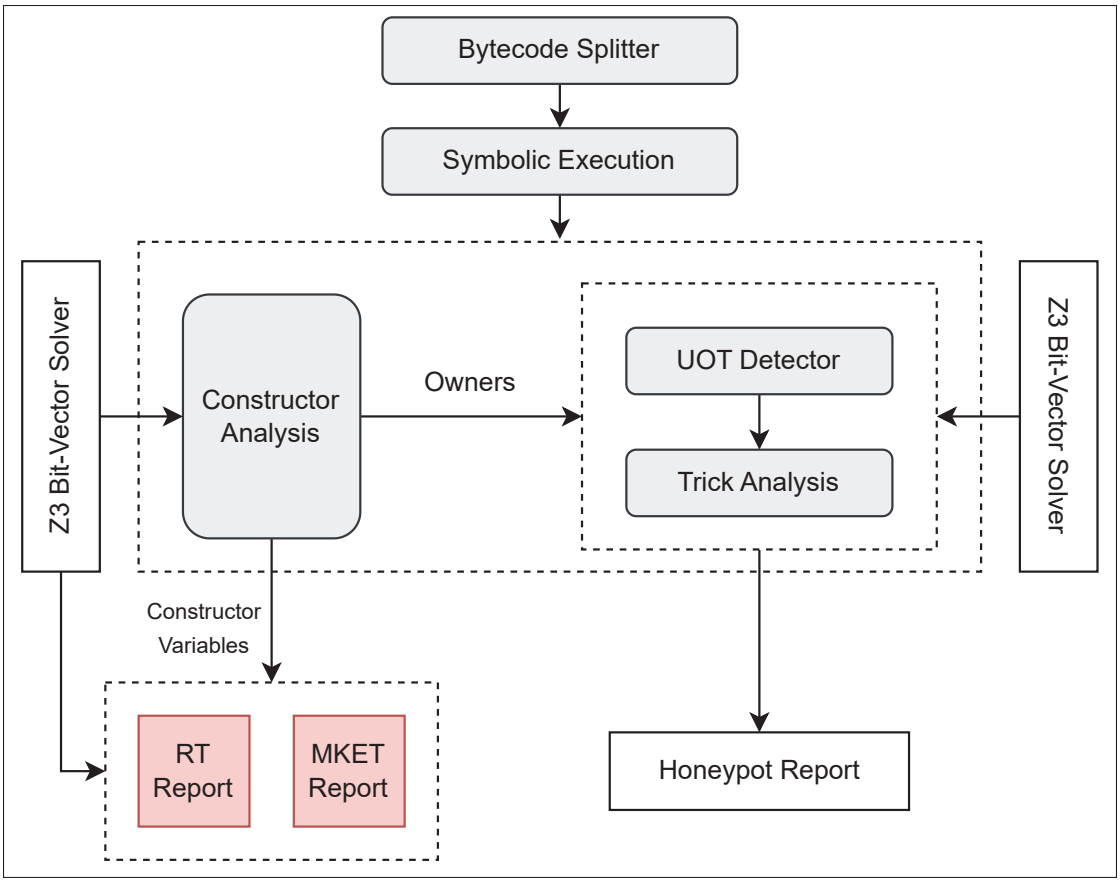

Figure 2.2 An overview of the HoneyVader's design showing the main components

detect racing time and map key encoding trick honeypots. As it is shown in the design overview, for each component we also use Z3 as the constraint solver.

2.4 Expanding Honeybadger

Other than our proposed method to detect honeypots, the constructor variables that are extracted during the constructor analysis process in HoneyVader can also be utilized to develop specific heuristics for detecting pre-defined honeypot techniques. The first heuristic aims to detect the racing time honeypot (explained in 1.4.3) and the second one focuses on the map key encoding trick honeypot (explained in 1.4.2). These heuristics are added to HoneyVader as two side

modules (shown in 2.2 in red boxes) which can then be integrated into Honeybadger's detection module as well.

The first heuristic uses the constructor variables to identify instances of racing time honeypots, which involve the use of time-based conditions to trick users into depositing money and withdrawing more but leaving no time for them to do the withdrawal transaction. By analyzing the path conditions of the contract calls, Honeybadger will be able to determine if a racing time honeypot is in play by checking for variables that are less than or greater than a certain timestamp value. The second heuristic is designed to detect map key encoding trick honeypots, which involve the manipulation of mapping variables to deceive users. By examining the list of hash values associated with the mapping variables and comparing them to the hash values used in the contract, Honeybadger will be able to identify if a map key encoding trick is being employed. The implementation of these two heuristics is more explained in 3.6.1 and 3.6.2. With the addition of these two heuristics, Honeybadger's detection capabilities are expanded, providing a more robust and comprehensive solution for detecting honeypots in Solidity contracts.

CHAPTER 3

IMPLEMENTATION

This chapter includes details about the implementation of our tool. Based on our design in Figure 2.2, our implementation is explained for each module. We implement our tool in Python with roughly 4,000 lines of code. In the following subsection, we describe the details of each component.

3.1 Splitting bytecode

The BytecodeSplitter module plays a crucial role in our methodology by extracting the bytecode related to the constructor parameters from the EVM creation bytecode. Creation bytecode which is already explained in 1.1.2.4 includes both the constructor and runtime parameters. The input to this module can be either the solidity source code or directly the EVM bytecode. In the case of solidity source code, in an additional step we compile it to the bytecode using `solc` compiler with the command `solc -bin` to get the binary of the creation bytecode in hex (shown in Figure 3.1). This creation bytecode is a sequence of hexadecimal numbers representing the opcodes

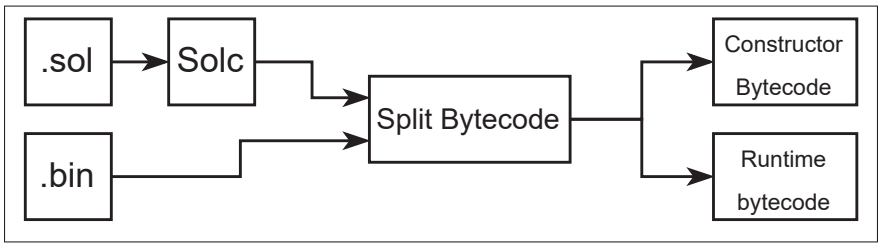

Figure 3.1 An overview of the structure of BytecodeSplitter module

and their corresponding arguments. The EVM uses these opcodes to execute the smart contract code. Each opcode has a specific function and is identified by a hexadecimal value. For example, the opcode "60" represents the PUSH1 instruction, which pushes a one-byte value onto the stack.

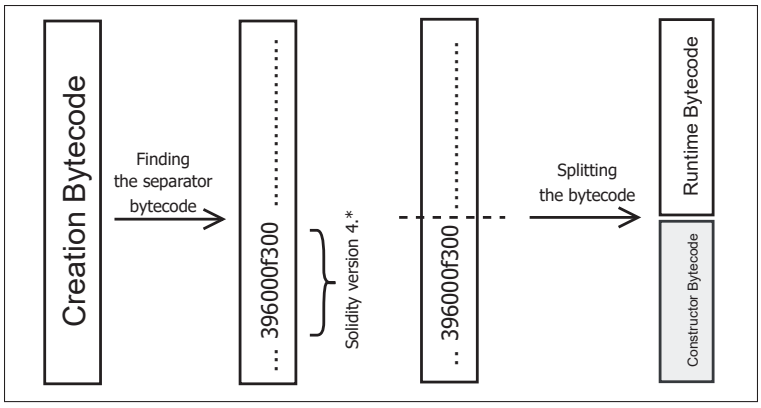

Figure 3.2 Splitting the EVM bytecode in a compiler version 0.4

The creation bytecode is a hexadecimal string that represents the compiled smart contract in its binary format and starts with the constructor bytecode followed by the runtime bytecode. Depending on the Solidity version, the constructor bytecode ends with a distinct set of opcodes that form the splitting line. Figure 3.2 depicts the bytecode splitting process for the compiler version 0.4. It ends with 396000f300 which represents the following instructions:

- **39 = CODECOPY:** This instruction is used when deploying a new contract to copy the bytecode from the contract creation transaction's data field to the contract's memory. The copied bytecode is then executed by the EVM to create a new instance of the contract.
- **6000 = PUSH1 0x00:** This instruction is used to push a 0 onto the top of the stack.
- **f3 = RETURN:** This instruction is used to return data from a contract after a function call. In this case, no data is specified and the length is zero, and this instruction will end the execution of the contract.
- **00 = STOP:** This instruction is used as a final instruction at the end of the contract's bytecode to prevent any further code from being executed. By executing this opcode the EVM immediately halts the execution of the current contract and returns any remaining gas back to the transaction's sender.

The mentioned instructions are executed at the end of the constructor parameters in the creation bytecode. However, different compiler versions perform different instructions. For the compiler

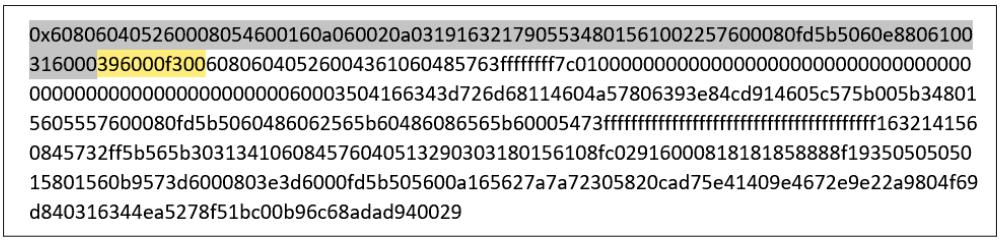

Figure 3.3 The creation bytecode of the contract deployed on Ethereum mainnet

versions more than 0.4 (i.e. 0.5 to 0.8) the bytecode related to the constructor ends with 396000f3fe (CODECOPY, PUSH1 0X00, RETURN, and INVALID) which is similar to the version 4, but a difference in the last instruction. Figure 3.3 shows the creation bytecode for a contract[3]. The piece of bytecode highlighted in yellow shows the end of the constructor section which is highlighted in gray. The rest is related to the runtime bytecode of the contract. We can simply realize that this contract is compiled using Solidity version 0.4.

3.2 Symbolic execution

In our tool, we have adopted and modified the symbolic analysis model introduced by (Luu *et al.*, 2016). This model symbolically executes the bytecode, explores all potential execution paths, and detects the constraints that must be satisfied for the program to behave in a particular manner. Oyente utilizes basic blocks to depict a sequence of instructions in the Ethereum Virtual Machine (EVM) code that are executed one after the other without any branches or jumps in between. For each block, it then evaluates if a path is feasible through the use of a Z3 constraint solver which takes a set of constraints as input and determines if there is any assignment of values to the variables that satisfies all the constraints. A feasible path refers to a specific execution path of a program that adheres to the constraints of the inputs and other specified conditions during the analysis. We define an opcode in a basic block satisfiable if there is a feasible execution path to it.

[3] Contract address: 0x223b0EE581719D4c6aE36f1BA1dd4101e5409c1c

As a result of the symbolic analysis performed on the constructor bytecode and runtime bytecode, we obtain useful information to guide the next steps. This information is separated for each bytecode and turns into the input for each module.

- **Constructor:**
  - A list of execution paths P.
  - A list of storage writes CON_S made during the SSTORE opcode, which is represented as a tuple $(con\_s_{var}, con\_s_{val})$ where con_s_{var} is the stored variable in the constructor, and con_s_{val} is the stored value in the constructor.
- **Runtime:**
  - A list of execution paths P.
  - A list of storage writes S produced during the SSTORE opcode represented as a tuple (s_{var}, s_{val}) where s_{var} is the stored variable and s_{val} is the stored value.
  - A list of calls C made during the CALL or DELEGATECALL opcode, represented as a tuple (c_r, c_v, c_t, c_{pc}) where c_r is the recipient, c_v is the call value, c_t is the type of call, and c_{pc} is the program counter of the opcode.
  - A list of suicides SC made during the SUICIDE or SELFDESTRUCT opcode, represented as a tuple (sc_r, sc_v) where sc_r is the recipient, and sc_v is the suicide value.

3.3 Constructor analysis

Smart contracts are deployed by an owner address, which is a unique public key associated with a wallet. Some contracts store the address of the creator in a constructor variable during the initial deployment transaction and are thus referred to as *owner-aware*. This allows us to retrieve information about the original contract creator. In contrast, other contracts do not store the address of the creator in any variables, making them *non-owner-aware*. This results in a lack of information about the original owner of the contract. After the completion of the symbolic analysis on the constructor bytecode, we can obtain information about the constructor parameters, including a list of owners, O. To identify the owners of the contract, we perform an iteration over all storage writes of the constructor, CON_S, and verify if there exists a storage

write, s, where $s_{val} == I_s$ (we prove this equation using Z3 solver). If such a storage write exists, the corresponding s_{var} is added to the list of owners, O. Finally, if O is not an empty set after the iteration, it indicates the detection of an owner-aware contract and goes to the next module. Otherwise, the contract is non-owner-aware and it will not continue preceding. The source code of the implementation to find the owner of a contract is appended in Listing II.1.

3.4 UOT detector

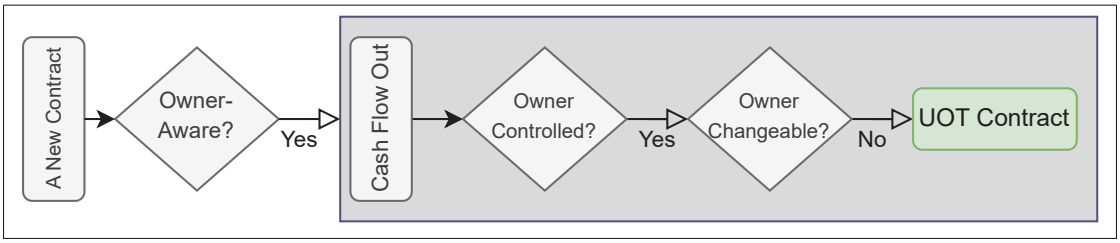

Figure 3.4 An overview of the workflow of UOT detector module

The overall overview of the UOT detector is shown in Figure 3.4. To detect a UOT contract, we iterate over all of the satisfiable calls C and suicides SC and check whether, for each call c (or sc), there exists an $owner$ contained in O who controls the cash flow out meaning: 1) $c_r == owner$ or $sc_r == owner$ proved by Z3 solver, or 2) c_r is a constant address meaning it is hard coded in the contract, or 3) the path conditions of c (or sc) contains a comparison between I_s and $owner$ meaning only $owner$ can trigger c (or sc), or 4) the path conditions of c contains a comparison between I_s and a constant address meaning a hard coded address by the contract's owner can trigger c (or sc).

If one of these conditions is true for c, we have to check if $owner$ is changeable in the code. To detect this, we iterate over all of the storage writes S and check whether there is a storage write s, where $s_{var} == owner$ proved by the Z3 solver. If $s \in \emptyset$, $owner$ is unchangeable and as a result of this stage, we add it to the list of funded unchangeable owners FUO. We also add c_{pc}

to the list of satisfiable calls to the owner SCO. The source code of the implementation to find out if the owner is changeable is appended to the appendix in Listing II.3.

After iterating over all of the calls contained in C and suicides contained in SC, we detect UOT, if $|SCO| = |C| + |SC|$. The source code to detect a UOT contract is appended in Listing II.2

3.5 Trick analysis

If the contract is a UOT contract, we proceed to identify two potential tricks in the contract:

- **Fake Money Transfer** We detect a fake money transfer by iterating through the list of calls C and checking if there exists a call c such that $c \notin SCO$, $c_v > 0$, and $c_r \neq owner$ for any $owner$ in O (Source code is appended in Listing II.5).
- **Fake Ownership Change** To detect any fraudulent ownership changes, we traverse the list of owners O and verify that for each $owner \in O$ there exists corresponding storage write s whose variable, s_{var}, is equal to $owner$, provided $owner \notin FUO$ (Source code is appended in Listing II.4).

3.6 Honeybadger's expansion

As is shown in the red box of Figure 2.2, we use the results of the constructor analysis to add two new modules to Honeybadger to detect new types of honeypots.

3.6.1 Racing time

To detect racing time in a smart contract, we first iterate over all the function calls contained in the contract and check if there exists a call c, whose path conditions consist of a comparison for the timestamp. We then collect a set of variables less than a specific timestamp value IH_s, and a set of variables greater than IH_s. Next, we examine the storage variables of the constructor and replace the variables corresponding to the less than and greater than sets with their respective values. Finally, we report a racing time if we find a comparison of lt (less than) and gt (greater than) such that $lt < IH\_s < gt$ and the difference between gt and lt is less than 5 minutes. This

method enables us to identify potential racing time vulnerabilities in the smart contract (Source code is appended in Listing II.7).

3.6.2 Map key encoding trick

In Solidity, a mapping is a key-value store that is commonly used to store variables. The hash of the map key is used as the address to store the variable in the mapping. However, this method of storing variables can be vulnerable to map key encoding tricks. To detect map key encoding tricks, we first iterate over a list of hashes contained in H. We check if there exists a hash h, whose value is used to store a variable both in the constructor and in the code. We then save the 'raw string' of the hash as a map key used in the contract. We also iterate over all the function calls contained in the contract C. We check if there exists a call c, where the map key m is used in its path conditions, for c_r or for c_v. We use the map key list to detect any instances of a map key encoding trick. Specifically, we look for a key in the map key list whose UTF-8 encoded form is the same as m, but has a different ASCII representation. If we find such an instance, we can flag it as a map key encoding trick (Source code is appended in Listing II.6).

CHAPTER 4

EVALUATION

In this chapter, we first present the parameters of our experiments and then propose our final results. Then we compare our results to the state-of-the-art tool Honeybadger which also uses symbolic execution for detecting honeypots. We end this chapter by analyzing one of the newly discovered honeypot techniques as a case study.

4.1 Experiments

In this section, we present the setup we used for the experiments and the dataset we used. The setup includes details on the hardware and software configurations of the machine used to run the experiments, as well as the parameters used for each experiment. The dataset includes information on the contracts deployed in Ethereum. By providing this information, we aim to provide a clear understanding of the experimental conditions and data used in our study.

4.1.1 Setup

For our experiments, we utilized a computing cluster consisting of two nodes. Each node was equipped with a single Intel Xeon Gold 5118 CPU, which boasts 48 cores operating at a frequency of 2.30GHz. The cluster was running on a 64-bit Linux kernel version 5.15.0-47-generic, which provided a stable and reliable environment for our experiments.

To obtain the initial bytecode of the contracts, we used the Solc compiler version 0.4.25, which is a popular Solidity compiler that compiles smart contracts into EVM bytecode. We then utilized version 1.8.16 of Geth's EVM to retrieve the assembly from the bytecode. This version of Geth's EVM was selected for its stability and compatibility with the version of Solc that we used. The combination of Solc and Geth's EVM provided a reliable and consistent way to generate the necessary bytecode and assembly for our experiments.

The symbolic execution setup included the following configurations:

- Z3 constraint solver version 4.7.1.
- Loop limit of 10.
- Depth limit for DFS of 50.
- Gas limit of 4 million.
- A timeout of 1 second per Z3 request.
- A timeout of 30 minutes to symbolically execute a contract.

Additionally, a timeout of 10 minutes was set to run heuristics for each contract. For convenience, we use Docker and create a container that contains the mentioned version of the dependencies, and then use that container to run the application in any environment that supports Docker.

4.1.2 Dataset

Our dataset is comprised of the creation bytecode of smart contracts deployed on Ethereum. To get the creation bytecode, we use web3.py (contributors, 2022) which is a Python library used to interact with the Ethereum blockchain. Connecting to an Ethereum node, allows the developers to send transactions, interact with smart contracts, and perform various other operations related to the Ethereum blockchain. To connect to an Ethereum node using web3.py, we need to access via an HTTP endpoint, which is typically a URL. The URL specifies the endpoint where the Ethereum node or provider is running and how to interact with it. In this project, we use Ankr as the blockchain provider to create an endpoint and make requests to it. Once we get access to the Ethereum blockchain, we can use different methods of web3.py to extract information from the blockchain. As the first step, we found all of the contract accounts (CAs) in the block range of our dataset by finding the transactions with an empty "to" field which specifies a CA (see paragraph 1.1.1.2). Moreover, for each contract, we extracted the creation bytecode from the input field of the first transaction information and stored the hash of the bytecode. Since over 98% of the contracts are duplicates, it was more efficient to store only their hashes instead of the whole bytecode. After eliminating duplicates, we removed contracts with similar bytecode

hashes, resulting in a set of 375,962 unique contracts. The initial bytecodes of these contracts were then downloaded and stored as our final dataset.

We conducted HoneyVader experiments on 375,962 unique smart contracts. The overall process took an average completion time of 323 seconds per contract. The results showed that 98% of contracts successfully completed the symbolic execution process and almost all of the contracts successfully completed the application of our heuristics in less than 10 minutes. Out of all the contracts evaluated, 121,876 contracts (32%) demonstrated a code coverage greater than 90%.

4.2 Metrics

In this study, we evaluate the effectiveness of our tool using several metrics.

- **TP (True Positives):** This category represents contracts that our methodology and verification confirm to be honeypots.
- **FP (False Positives):** This group includes the contracts that were flagged as honeypots by our tool but are not verified honeypots in the verification phase.
- **FN (False Negatives):** This class of contracts refers to the ones that were detected as benign contracts by mistake but are actually honeypots. We use this metric to compare HoneyVader to Honeybadger.
- **Precision:** P is the proportion of true positives among all predicted positives and we use it to evaluate our detection tool. It is calculated as:

$$p = TP/(TP + FP) \tag{4.1}$$

4.3 Results

In this section, we present the results of our analysis based on the metrics defined in section 4.2. Our study successfully extracted constructor parameters for all the contracts in the bytecode dataset. Specifically, out of the analyzed contracts, we identified 16,687 that were owner-aware, and these were further analyzed for potential honeypot activity. From this group, we uncovered

4,399 UOT contracts that accepted funding from any source but transferred it to only one address, which was invariably the owner's address. Given that such behavior is indicative of potential honeypot activity, we considered these contracts suspicious and marked them for further investigation. Overall, HoneyVader labeled 139 contracts as honeypots. From this

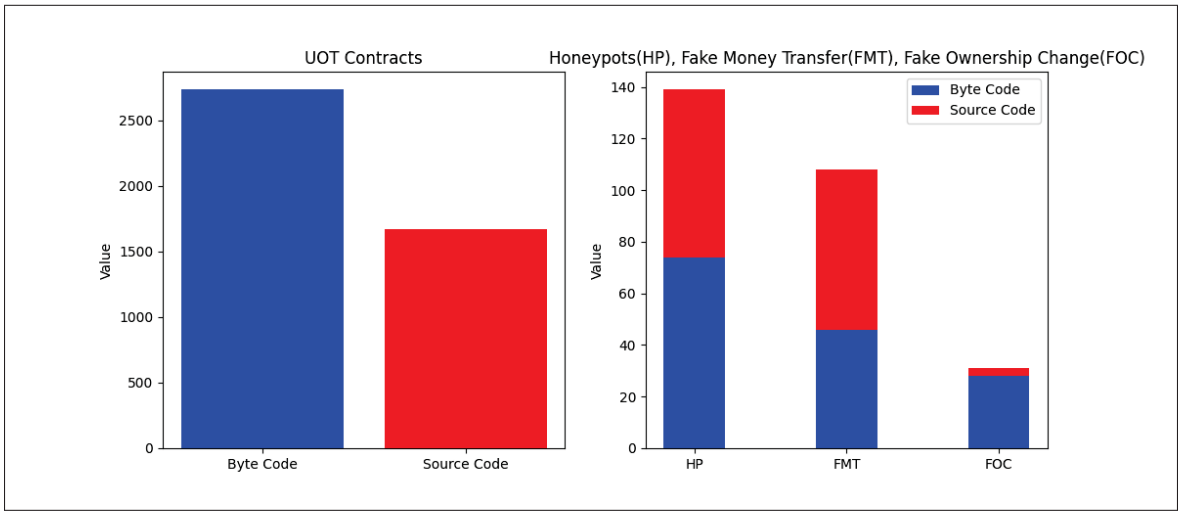

Figure 4.1 Number of contracts with source code and bytecode for UOT contracts, honeypots using fake money transfer, honeypots using fake ownership change and honeypots overall

number 65 of them are open-source and the rest have only their bytecode available. Our analysis has revealed that 108 of the suspicious contracts employ a fake money transfer method to deceive victims, while 31 entice them into taking ownership of the contract with the hope of withdrawing money. These results emphasize the critical importance of exercising caution when interacting with contracts that exhibit the UOT characteristic. To visually illustrate our primary findings, we have included a figure that clearly displays the frequency of UOT contracts and each type of attack method used by attackers(see Figure 4.1). Interestingly, our results indicate that attackers tend to rely more on the fake money transfer method, as opposed to the fake ownership change, in their honeypot schemes.

Transaction Hash	Method	Block	Age	From		To	Value	Txn Fee
0x9eb19ed4b0867bf99...	Close	7215349	1498 days 17 hrs ago	0x310Bac...cDb0c5FF	IN	0x223b0E...e5409c1c	0 ETH	0.00005325
0x94a086165fb36ed45...	Play	7213961	1499 days 1 hr ago	0xA48AC3...881E8a57	IN	0x223b0E...e5409c1c	3.1 ETH	0.00017456
0x05a32251426d7da8...	Transfer	7213591	1499 days 3 hrs ago	0x3B0313...532F00af	IN	0x223b0E...e5409c1c	0 ETH	0.00016832
0xbf7d62179d722f80f...	Transfer	7213233	1499 days 5 hrs ago	0x310Bac...cDb0c5FF	IN	0x223b0E...e5409c1c	3.000001 ETH	0.00004208
0x91a1c762ece7ff9e1...	0x60806040	7213214	1499 days 5 hrs ago	0x310Bac...cDb0c5FF	IN	Create: HumpDayPlay	0 ETH	0.00027203

Figure 4.2 Normal transactions of the contract

Parent Txn Hash	Block	Age	From		To	Value
0x9eb19ed4b0867bf9...	7215349	1456 days 12 hrs ago	0x223b0E...e5409c1c		0x310Bac...cDb0c5FF	6.100001 ETH

Figure 4.3 Internal transactions of the contract

4.4 Validation

To validate our results about honeypot contracts, we conducted a thorough manual analysis of the detected honeypots with their source code available. Our validation process consisted of both dynamic and manual analysis.

As it is mentioned in 4.3, 65 contracts detected as honeypots have their source code available. For our first step of validation, we used Etherscan (Etherscan, n.d.) to obtain the source code of each one of these contracts and manually analyzed the code for any potential honeypot tricks. To verify our manual analysis, we then ran each contract in Remix IDE (Remix IDE, n.d.) and performed transactions in a test environment to observe the contract's behavior.

In addition, we utilized transaction behaviors as a form of validation for these 65 contracts. Honeypots that were successful in trapping victims show a rug pull pattern in their transactions, where they wait for a victim to deposit funds before pulling all the funds out of the contract. Along with normal transactions that show ETH transfers carried out through a smart contract, we also analyzed internal transactions, which show the outgoing funds of the contract triggered by an external transaction. These transactions reveal the accounts to which the funds have been transferred.

To make this pattern clear, we show a list of normal and internal transactions of a honeypot in Figure 4.2 and Figure 4.3.[4] When two users are caught in the honeypot and lose their money, the attacker calls the function `close` to withdraw all of the money from the contract. Moreover, the internal transactions prove that there is only one user who makes money from the contract, and no other users could benefit from the contract.

Finally, we conducted a comprehensive review of the comment section on Etherscan, which often includes valuable information and warnings about scams or honeypots. This enabled us to gather additional insight into the potential threats posed by the contracts we analyzed. In particular, we searched for comments that mentioned issues or red flags associated with the contracts, such as suspicious or fraudulent behavior. Additionally, this helped us ensure the accuracy and reliability of our findings, as we were able to confirm the existence of certain contracts and their associated risks, based on the feedback and experiences shared by other users.

Table 4.1 Number of the evaluation set, owner-aware contracts, UOT contracts, true positives and false positives for each honeypot tricks

Evaluation Set	**Owner Aware**	**UOT**	**Honeypot**			
			Fake Money Transfer		**Fake Ownership Change**	
375,962	16,687	4,399	TP	FP	TP	FP
			57	5	2	1

The verification of our results is summarized in Table 4.1. Overall out of 4,399 suspicious UOT contracts, we detect 65 honeypots. From this number, 59 of them are true positives as well as 6 false positives which happened as the result of our limitations explained in subsection 5. Overall, according to Table 4.1 and the equation 4.1, our tool achieves a 90% precision in detecting intrinsic honeypots, regardless of the technique used in them. The results of our study also reveal some new honeypots that were not detected in previous works. These honeypots referred to as zero-day honeypots are listed in Table III-1 in the Appendix. These newly discovered honeypots demonstrate the need for continued research and development in the field of honeypot detection to stay ahead of emerging threats.

[4] Honeypot address: 0x223b0EE581719D4c6aE36f1BA1dd4101e5409c1c

4.5 Baseline solution

Table 4.2 Comparing Honeybadger's results to HoneyVader

	Honeybadger		**HoneyVader**				
	TP	FP	Owner Aware	UOT		Trick Found	
				TP	FP	TP	FP
BD	20	-	20	19	-	19	-
ID	41	7	47	43	1	-	-
SESL	12	-	12	-	-	-	-
TDO	4	-	4	4	-	4	-
US	32	-	31	17	-	16	-
HT	12	-	12	-	-	-	-

In order to evaluate the performance of our honeypot detection tool, we selected Honeybadger as a baseline model. Honeybadger is a widely recognized tool for honeypot detection and employs the same analysis methodology as our tool. By comparing HoneyVader to Honeybadger, we can better understand the strengths and weaknesses of our approach. To conduct the comparison, we performed two tests. The first test involved analyzing the baseline dataset of Honeybadger's results separately for each intrinsic honeypot technique. Since Honeybadger presents its performance results separately for each honeypot technique, we discussed the comparison for each technique individually. We also summarized our results in Table 4.2, which provides a comprehensive overview of the performance comparison between our tool and the Honeybadger baseline.

- **Balance Disorder:** Our results indicate that our tool successfully identified all 19 UOT contracts in the balance disorder dataset, with the exception of one that could not be detected due to the limitations of the Z3 solver. Additionally, we detected one zero-day honeypot that utilized a balance disorder technique.
- **Inheritance Disorder:** Regarding the inheritance disorder dataset, our tool found 43 UOT contracts among the 48 honeypots detected by Honeybadger. Combining our UOT heuristic with Honeybadger's inheritance disorder heuristic showed an increase of 6% in precision. To make this combination, we first ran the UOT detector on all 48 honeypots flagged by Honeybadger and detected 44 of them as UOT. Then we ran the ID heuristic on the 44 UOT

contracts resulting in 44 flagged honeypots. From this number, 4 of them were false positives while 40 of them were detected as true positives. We calculate the precision of Honeybadger and (HoneyVader + Honeybadger) as follows:

- **Precision of Honeybadger:** 41 / (41 + 7) = 0.85
- **Precision of (Honeybadger + HoneyVader):** 40 / (40 + 4) = 0.91

- **Skip Empty String Literal - Hidden Transfer:** Unfortunately, due to limitations of our symbolic execution engine in handling `throw` in Solidity, we were unable to detect any UOT contracts in the skip empty string literal - hidden transfer dataset using HoneyVader. However, our tool discovered three new zero-day honeypots with a hidden transfer technique that were not detected by Honeybadger.
- **Uninitialized Struct:** In the uninitialized struct dataset, we identified 17 UOT contracts among 31 owner-aware contracts. After validating the non-UOT contracts, we discovered that 13 of the contracts detected as true positive by Honeybadger were not honeypots, and in five of them, we observed users withdrawing money from the contract. This highlights using UOT heuristic before trying to find honeypots. Moreover, one of the detected honeypots also did not utilize the uninitialized struct technique, highlighting the importance of detection tools being independent of specific honeypot techniques. Lastly, we found one zero-day honeypot in this dataset.

During the second round of experiments, we applied HoneyVader to all the datasets used by Honeybadger, which included 151,935 unique contracts. Within this dataset, we discovered 7 instances of false negatives that were missed by Honeybadger. Out of these, 5 contracts used techniques previously identified in (Torres *et al.*, 2019), while the remaining 2 employed a new technique. However, the specific technique used is not our primary concern; what matters most is whether the contract is UOT and capable of deceiving the user. Table III-1 contains the addresses of these honeypots.

4.6 Case study

In this section, with the help of a real-world honeypot contract from our results, we explain the various steps of our methodology. This honeypot shown in 4.1 is a new type that is not using any of the defined techniques in (Torres *et al.*, 2019). The malicious contract operates by using multiple comparisons and assignments in its code to confuse users. An unsuspecting user may assume they can withdraw funds from the variable `ownerTax` by invoking the `AddTicket` function. Using HoneyVader, we first identify this contract as owner-aware by analyzing the constructor variable, which stores `msg.sender`. We found the variable `owner` in slot 0 of storage. Next, we analyze the contract's cash flow using runtime bytecode analysis. The money transfers in this contract include `suicide` in line 15 and two transfers in line 29. Analyzing all the feasible paths from the symbolic execution results, only the `suicide` is possibly happening in this contract which is in control of the owner (`onlyOwner` modified is used). Analyzing the storage writes in the contract, we do not find any new owner assignment and therefore the owner of the contract is unchangeable. These characteristics make the contract a UOT contract and lead us to the next step of the analysis which is looking for a trick. In this step, we look for a money transfer in the code that exists but never executes. By analyzing all of the money transfers we find the ones in line 29 impossible to happen, meaning there are no feasible paths to them when simulating the execution with symbolic variables. These fake money transfers in a UOT contract report this contract as a honeypot. Further analysis of the source code also reveals that these money transfers are intentionally included to deceive users into depositing money into the contract and prevent them from withdrawing funds.

```
contract Owned {
    address owner = msg.sender;
    modifier onlyOwner() {
        assert(msg.sender == owner);
        _;
    }
}
contract NewLottery is Owned {
    uint256 private maxTickets;
    uint256 numtickets;
    uint256 totalBounty;
    address owner;
    ..
    function shutdown() onlyOwner public {
        suicide(msg.sender);
    }
    function AddTicket() public payable {
        require(msg.value == ticketPrice);
        require(numtickets < maxTickets);
        bool success = numtickets == maxTickets;
        if(success)
            PayWinner(msg.sender);
    }
    function PayWinner( address winner ) private {
        require(numtickets == maxTickets);
        uint ownerTax = 5 * totalBounty / 100;
        uint winnerPrice = totalBounty - ownerTax;
        ...
        owner.transfer(ownerTax);  winner.transfer(winnerPrice);
    }
}
```

Listing 4.1: A simplified example of a zero-day discovered honeypot

CHAPTER 5

LIMITATIONS

Symbolic execution is an effective method for analyzing the behavior of smart contracts. However, there are challenges associated with our use of this technique in detecting UOT contracts. In order to accurately identify all potential transfers out of a contract, it is crucial to explore every possible execution path within a contract. However, in some cases, particularly with complex and extensive contracts, symbolic execution may not be able to cover all paths, leading to missed fund transfers and generating false UOT contracts. This can result in false positives and decrease the precision of our tool.

Another challenge in the analysis of smart contracts using our tool is the reliance on bytecode representation. Despite the advantage of detecting honeypots based on bytecode, the limited availability of open-source contracts (only 2% of all contracts) makes it difficult to identify certain types of honeypots. Some information is lost during the compilation process from source code to bytecode, such as operations that are present in the source code but do not change the state of the contract. For instance, a newly proposed honeypot technique, unexecuted call honeypot (Camino *et al.*, 2020), involves the attacker performing a fund transfer through a call without the necessary parenthesis to execute the call. This results in the call not appearing in the bytecode and only being visible in the source code. Our tool can still detect this contract as UOT since it is an intrinsic honeypot and this is considered a strong point in HoneyVader. However, finding the trick in such contracts can be difficult, as the bytecode does not provide any information about the call.

Finally, it is important to note that calls in smart contracts can take the form of either ETH transfers or calls to other contracts. While they may appear distinct in the source code, EVM treats them similarly and does not differentiate between them in the bytecode. In order to accurately detect UOT contracts, it is necessary to distinguish between ETH transfers and contract calls. This can also lead to false positive results and reduce the precision of our analysis tool.

CONCLUSION AND RECOMMENDATIONS

In this thesis, we have presented a novel approach to detect honeypot scams in Ethereum smart contracts at the bytecode level through symbolic execution. Our findings demonstrate that analyzing creation bytecode instead of runtime bytecode provides valuable information about constructor parameters, enabling the development of a heuristic to detect honeypots without prior knowledge of the specific trick used. Our approach was able to detect different types of honeypots based on the Honeybadger classification and identified a new class of honeypots. Moreover, we expand Honeybadger to detect more honeypots. In conclusion, our approach to detecting honeypot scams in Ethereum smart contracts offers a significant contribution to the field and provides a valuable tool for protecting against malicious actors. In the next section, we give a short explanation of the lessons we learned through this research.

Lessons learned

In this section, we provide a retrospective analysis of the research process, highlighting key insights gained through the study. We also discuss some challenges we encountered during the research, the methods and approaches that worked well, and those that did not. Giving a comprehensive overview of the research process, we explain what we learned, what we could have done differently, and what recommendations we would make for future research in this field.

Analysis choices

A comprehensive literature review, coupled with adequate technical skills, enables researchers to make the right choices regarding their research materials. While it is time-efficient and beneficial to take inspiration from previous research, following the exact path of others may not always be the best choice for one's own research. In our thesis, due to limited time and experience, we

opted to use Honeybadger's symbolic execution engine, which was taken from Oyente (Luu *et al.*, 2016). While this engine produced impressive results for Honeybadger, it was not the optimal choice for our tool's implementation, and we had to make manual changes to improve performance, leading to frustration and delays at the last phase of the research, evaluation. After facing these challenges, we realized that implementing our algorithm with another symbolic execution engine, such as Manticore (Mossberg *et al.*, 2019), may have been a better choice as the information it provides is more useful for our methodology. In conclusion, while analyzing the methodology used in related work can be beneficial, researchers should consider other options and have a strong understanding of their project's needs to choose the most efficient analysis methodology during the evaluation phase.

Large amount of data

When dealing with a large amount of data, proper classification and organization of data are crucial, as the risk of confusion or data loss is high, and it can impact the research results. In our thesis, our dataset comprised over 2,000,000 smart contracts downloaded from the Ethereum blockchain, which presented a significant challenge. We learned three crucial lessons during this phase of our research, which we will discuss below:

- **Data Classification:** Data classification was a critical aspect of our research as it enabled us to organize and manage our vast dataset. To accomplish this, we classified the contracts into specific groups based on their block range and kept track of the number of contracts in each class. This step was crucial as even the loss of a single contract by mistake could have adversely impacted the performance of our detection tool. By organizing the data into different categories, we were able to minimize the risk of confusion and data loss. The process of classifying the contracts also made it easier to restore a lost contract as we had information about its block number. Additionally, by keeping track of the number of contracts in each class, we were able to ensure that the number of contracts in each group remained

consistent throughout the research. Any discrepancies in the number of contracts could have significantly affected the reliability and validity of our results.

- **Data Storing:** In our research, we worked with a massive dataset of smart contract bytecodes. However, since we only needed to test our methodology on unique contracts, storing duplicates was unnecessary. We found that only 2% of the entire dataset were unique. As a result, we decided to store only the hashes of the contracts, which occupied much less space, and then we removed duplicates based on their hash. This approach enabled us to significantly reduce the amount of space needed for storage. It is essential to note that storing such large datasets can be done more efficiently than directly downloading them onto a local or cluster computer. Additionally, it is crucial to have a backup of the stored dataset in case of any unexpected events such as hard drive formatting or data corruption.
- **Data Process:** When working with large datasets, it is very important to approach data processing with a well-planned strategy. In our research, we got wrong in simply running our methodology on the dataset and waiting for the results which took about 10 days. It was only at the end of this process that we discovered bugs and errors that could have affected our results, and we had to start the whole process again. We also identified false positives at the end of the execution, which helped us understand the flaws in the methodology and improve its performance. This required us to fix certain aspects of the methodology and then run it again. We learned an important lesson from this experience: testing a project with a dataset of 10 samples is not sufficient to detect all errors and produce accurate results. Thus, when working on a large project with a large dataset, it is essential to have a representative sample size that is large enough to present the efficiency of the methodology and detect bugs, but small enough to avoid prolonged processing times. This approach can prevent researchers from repeatedly running the project on a large dataset, which can be time-consuming and challenging.

Future directions and recommendations

In this thesis, we focused on detecting Ethereum honeypots, but we believe that this is just the beginning of research on honeypots in the blockchain. Our investigations led us to identify a new type of scam in blockchain called honeypot coins. These scams target naive cryptocurrency users by enticing them to buy coins from a contract, waiting for others to buy in, and then making it impossible for users to cash out their investments, except for the contract owner. Honeypot coins are particularly prevalent in the Binance Smart Chain (BSC), and detecting them is challenging as they may appear to be a normal newly deployed coin gaining value on a chart.

We believe that honeypot coins share characteristics with the honeypots we studied in this research, and our proposed methodology can be a practical research direction for future researchers in this field. To the best of our knowledge, no academic research has focused on this area, and there is no prominent detection tool to detect honeypot coins. As such, there is an opportunity for further research and development in this area, and we hope our work can inspire and guide future efforts to confront honeypots in cryptocurrency scams.

APPENDIX I

THE GREED TRAP: UNCOVERING INTRINSIC ETHEREUM HONEYPOTS THROUGH SYMBOLIC EXECUTION

Mahtab Norouzi , Mounir Elgharabawy , Kaiwen Zhang

Department of Software Engineering and IT, École de technologie supérieure,
1100 Notre-Dame St W, Montreal, Quebec H3C 1K3

Paper is submitted to the conference «Blockchain Computing and Applications (BCCA 2023, Kuwait City, Kuwait)»

1. Abstract

Smart contracts are self-executing computer programs that run on blockchain networks and facilitate secure, transparent, decentralized transactions. The security of smart contracts has always been a critical issue in the blockchain community, and one of the major concerns in recent years is the prevalence of honeypots – malicious contracts designed to deceive users into depositing funds, only to find that they are unable to withdraw their money and have lost their original deposit. In this research, we present a novel classification of honeypots and introduce a new type of contract that enables the development of a future-proof method for detecting honeypots based on the contract owner and cash flow. We implement this method in the form of a detection tool called HoneyVader using symbolic execution to identify real-world honeypot contracts. We apply our tool to over 2 million contracts deployed on the Ethereum network and detect 139 honeypots. Using this tool, we uncover zero-day honeypots and new techniques used by attackers, in addition to the ones identified in previous works.

2. Introduction

Blockchain is a publicly shared immutable ledger between nodes in a peer-to-peer network that allows the users to interact directly in the absence of any third party in a trustless environment. Blockchain has emerged as one of the most prominent technologies in recent years. Alongside the potential for decentralization and transparency, this technology has had a significant impact

on many industries, ranging from technology to the healthcare industry and business. As the first version of the blockchain, Bitcoin cryptocurrency (BTC) was introduced in 2009 (Nakamoto, 2009), gaining a lot of attention, and the technology rapidly grew to the next version in just a few years. In 2015, Ethereum was launched as blockchain 2.0, allowing programmers to develop codes called smart contracts in the blockchain in addition to exchanging cryptocurrencies (Wood, 2014). The critical characteristic of smart contracts is their immutability, which means they cannot be altered after deployment. Smart contracts can be developed by a variety of programming languages such as Solidity (Solidity, 2016) Vyper (Ethereum Foundation, 2021), Bamboo (Serrano & Clack, 2018). However, Solidity is the most popular for programmers to work with.

As blockchain is growing more in different industries, security is also becoming more of an issue for users. Due to the massive amount of money flowing on the chain and the irreversible nature of transactions, any vulnerability in smart contracts could lead to huge losses. One of the most famous ones, the reentrancy attack happened in 2016 which caused the hack of DOA and the stealing of 3.6 million Ether (Buterin, 2016). However, the security of the blockchain is not always about detecting vulnerabilities of smart contracts. The pseudonymous environment in the blockchain makes it an attractive target for fraudsters and scammers. In recent years, several scams have been reported in blockchain, including rug pulls, Ponzi schemes, and honeypots, particularly in decentralized finance (DeFi) where cryptocurrency plays a larger role.

Honeypots are currently a major issue in the blockchain. These malicious contracts present a code leak and lure users into depositing and withdrawing more money than their initial stake (Torres *et al.*, 2019). However, after sending the money to the contract, it gets stuck and the victim will not be able to withdraw any funds. Finally, the attacker drains the contract balance to their address.

There are some studies conducted to detect these honeypots. (Torres *et al.*, 2019) was the first study to introduce honeypots and classify them into 8 categories based on the used technique by the attacker to trick the users. They also introduced Honeybadger, the most prominent tool for

honeypot detection based on symbolic execution on the EVM bytecode. However, Honeybadger is limited to detect only 8 types of honeypots studied by the authors, and since new honeypots are continuously proposed (Camino *et al.*, 2020; Zhou *et al.*, 2020), there is a need to develop a general purpose solution that is able to detect honeypots even the ones that are not previously known. Honeybadger uses runtime bytecode to analyze a contract. However, the constructor of a contract, which is placed in the creation bytecode, contains important information that can be useful in the analysis of the contract for honeypot detection. The creation bytecode which also generates the runtime bytecode and carries the constructor information is generated only once when a user creates and deploys a contract to the blockchain.

In this paper, we present a novel approach for detecting honeypots by conducting symbolic analysis on the EVM creation bytecode. We introduce a new classification system for honeypots, Intrinsic and Non-intrinsic. Based on this classification and by analyzing the constructor parameters of a contract, we develop a heuristic based on the cash flow of a smart contract that goes beyond the techniques introduced by prior work, including the Honeybadger tool. Our method identifies Unique Owner Transfer (UOT) contracts as potentially malicious ones, adding an additional layer of detection to check if they are benign contracts or honeypots. With these strategies, we can detect honeypots even if we are not familiar with the specific techniques used to create them. In addition, we leverage our constructor analysis to develop two new heuristics for detecting honeypots recently introduced in other studies, which we incorporate into the Honeybadger tool.

Our research methodology is based on extracting information about the owner of a contract. Specifically, our tool is designed to identify honeypots in Ethereum contracts where the message sender is assigned to a variable in the constructor, as this variable represents the contract owner. Contracts that do not explicitly define the owner of the contract, which we refer to as non-owner-aware contracts, are not within the scope of our research. Additionally, our objective in this research is to identify honeypots as standalone entities, disregarding any extrinsic interactions or transactions. Contracts that utilize trickery dependent on external factors or other contracts cannot be fully analyzed by our tool, as these features are not inherent within the contract's code.

Therefore, our research is limited to detecting only the intrinsic honeypots, and non-intrinsic honeypots are out of the scope of our tool. Our main contributions are as follows:

- We propose a new classification for honeypots with a new perspective based on their life cycle.
- We propose a method to find the constructor variables of a contract in order to find the owner variable of a contract.
- We propose a novel future-proof method able to detect unrecognized forms of honeypots that are not covered in existing heuristics.
- We implement our solution, HoneyVader, and evaluate it using a dataset of 2 million Ethereum smart contracts against Honeybadger, a well-known competitor.
- we expand the range of honeypot detection in Honeybadger from 8 to 10.

The remainder of this paper is organized as follows. In section 3, we investigate some of the previous works done related to our research area. Section 4 explains some principles of blockchain and some background on our analysis methodology. Section 5 presents our main idea to detect honeypots and an overview of our detection tool. In section 6, we demonstrate our methodology and the implementation of our detection tool, HoneyVader in more detail and evaluate it in section 7 using a large dataset of smart contracts. We explain some limitations of our research in section 8, and finally, conclude our proposed method in the last section.

3. Related work

The advancement of decentralized applications (dApps) and decentralized finance (DeFi) has greatly increased the focus on security in the blockchain space. Researchers have been focused on detecting vulnerabilities in smart contracts and exploring related areas such as fraud detection. A systematic review of 86 smart contract analysis tools was conducted by (Tama *et al.*, 2017a). In addition to vulnerabilities, researchers have also studied Ponzi scams and phishing attacks, developing detection tools to help prevent these types of frauds. Studies such as (Bartoletti *et al.*, 2020-01, 2018-06; Chen *et al.*, 2018; Vasek & Moore, 2018) have focused on Ponzi scams, while

(Abdelhamid *et al.*, 2014; Chen *et al.*, 2020-07; Wen *et al.*, 2021) have focused on identifying phishing attacks to provide a safer environment for blockchain users.

Our main focus in this paper is a relatively new type of fraud, honeypots. During the time of this study, the following works are done to uncover this scam on Ethereum. Static analysis methodologies, such as symbolic execution and Control Flow Graph (CFG), facilitate a more precise path-by-path analysis of the program compared to dynamic analysis methods. (Torres *et al.*, 2019) explores various types of honeypots and proposes a taxonomy based on attacker techniques. Honeybadger uses 8 heuristics combined with symbolic execution, which can be expanded to detect new honeypot techniques that are defined and analyzed by experts. We use this feature in our work and add two new heuristics for honeypots defined in (Zhou *et al.*, 2020) and (Camino *et al.*, 2020) (i.e. Racing Time (RT) and Map Key Encoding Trick (MKET)). However, it is worth noting that the tool is limited to recognizing disclosed techniques that have been analyzed by experts and cannot detect new techniques that have not yet been discovered.

Data science approaches, including the use of machine learning algorithms, have also been employed for honeypot detection in Ethereum. (Camino *et al.*, 2020) utilizes the XGBoost machine learning algorithm to classify Ethereum smart contracts as honeypots and non-honeypots by extracting features from source code, transaction data, and flow of funds. However, relying solely on source code led to excluding over 98% of all Ethereum smart contracts that were not open-source, and using transactions as a feature resulted in excluding over half of the honeypots that had zero transactions.

Subsequently, some studies aimed to incorporate Solidity opcodes in the feature extraction process such as (Hara *et al.*, 2021) since analyzing contracts based on bytecode is crucial. However, this approach still faces the challenge of an imbalanced dataset, where the number of negative samples (non-honeypots) is much larger than positive ones (honeypots), which hinders the classification capabilities of machine learning algorithms.

There are other studies with different primary objectives besides identifying honeypots in smart contracts. In (Bian *et al.*, 2021), the authors use code visualization to generate a model based on

RGB images of Solidity bytecode and ABI, with the primary goal of detecting Ponzi schemes. The performance of the model in detecting honeypots is also evaluated using the results from Honeybadger. In (Zhou *et al.*, 2020), the authors propose a method to analyze real-world attacks by examining contract transactions, allowing them to discover new types of honeypots along with previously unknown zero-day attacks.

In this study, we take advantage of the power of symbolic execution at the bytecode level to detect honeypots. Unlike some approaches, our method does not require a large amount of data to be effective. Instead, we propose a new way to detect honeypots that can be applied regardless of the specific technique used by the attacker, transactions, or the availability of source code. This enables us to detect honeypots whose techniques are not yet known. Moreover, some honeypots do not use any specific technique, especially the ones without source code because the victim cannot see the source and fall into a trap.

4. Background

In this section, we explain some of the fundamentals of blockchain and smart contracts as well as analysis methodologies.

4.1 Smart Contracts

Smart contracts are self-executing computer programs that run on blockchain networks. They run on decentralized blockchain networks, allowing for a tamper-proof and transparent execution of their code without the need for intermediaries. In recent years, they have gained widespread attention for their potential to revolutionize various industries by making direct interactions possible and reducing costs. Smart contracts have the ability to automate complex transactions, enforce contractual agreements, and store data securely, making them a promising tool for improving efficiency, security, and trust in digital transactions.

The data stored in the blockchain, including smart contracts, is publicly available, making it transparent and accessible to all network participants. Despite this public accessibility, smart

contracts are not always open source. Instead, the runtime bytecode, which represents the compiled version of the smart contract, is stored on the blockchain. This bytecode contains all the information necessary for the execution of the contract on the network. In many cases, users choose to keep the source code private and only the compiled bytecode is visible on the blockchain. While some users do publish their contract source code on block explorers, this remains a relatively uncommon occurrence.

4.2 EVM and bytecodes

Ethereum Virtual Machine (EVM) is a crucial component of the Ethereum blockchain. It is a 256-bit word, a stack-based machine that acts as a virtual CPU capable of executing smart contracts. The EVM is designed to provide a secure and decentralized platform for executing code on the blockchain. By using the EVM, developers can write and deploy smart contracts that can be executed by nodes on the network. The EVM is responsible for maintaining the state of the blockchain, including account balances, contract codes, and storage. It reads bytecode, a low-level programming language, and executes it on behalf of the nodes in the network. Geth, one of the implementations of Ethereum, includes the EVM and is run by all nodes in the Ethereum network. The EVM's design makes it possible for smart contracts to execute in an environment that ensures the safety of the network.

After a smart contract written in Solidity is compiled, it is translated into a low-level, binary representation that is readable by EVM, known as EVM bytecode. EVM bytecode defines the behavior of the smart contract, including how it stores and manipulates data, interacts with other contracts, and processes transactions that are stored on the blockchain as part of the smart contract, where it is publicly visible and immutable. This means that the behavior of the smart contract is fully transparent and predictable.

There are two types of EVM bytecode:

- **Creation Bytecode:** The creation bytecode also known as the initial bytecode, is a crucial component in the deployment of smart contracts on the EVM. It is executed only once during

the creation transaction and generates the runtime bytecode. The creation bytecode contains the information necessary for the constructor and is used to write initial variables to the contract's storage. This bytecode must be included as part of the deployment process, which involves the submission of a transaction from the creator's address to the null address.

- **Runtime Bytecode:** The runtime bytecode refers to the part of the creation bytecode that executes in each subsequent transaction, excluding the constructor logic and parameters included in the initial creation transaction.

4.3 Smart contracts

Smart contracts are self-executing computer programs that run on blockchain networks. They run on decentralized blockchain networks, allowing for a tamper-proof and transparent execution of their code without the need for intermediaries. In recent years, they have gained widespread attention for their potential to revolutionize various industries by making direct interactions possible and reducing costs. Smart contracts have the ability to automate complex transactions, digital agreements between parties, and store data securely, making them a promising tool for improving efficiency, security, and trust in digital transactions.

The data stored in the blockchain, including smart contracts, is publicly available, making it transparent and accessible to all network participants. Despite this public accessibility, smart contracts are not always open source. Instead, the runtime bytecode, which represents the compiled version of the smart contract, is stored on the blockchain. This bytecode contains all the information necessary for the execution of the contract on the network. In many cases, users choose to keep the source code private and only the compiled bytecode is visible on the blockchain. While some users do publish their contract source code on block explorers, this remains a relatively uncommon occurrence.

4.4 Honeypots

The phenomenon of honeypots in Ethereum is a type of scam that involves luring greedy users into depositing funds into a malicious smart contract (Torres *et al.*, 2019). The victim is enticed

by the promise of being able to withdraw more funds than they initially deposited, either through a deliberate feature in the code or by exploiting a vulnerability or error made by the programmer. To further entice the victim, a small amount of funds is placed as bait within the contract. However, once the deposit is made, the funds become trapped within the contract, rendering them permanently inaccessible to the victim.

According to the taxonomy proposed in (Torres *et al.*, 2019), honeypots in Ethereum can be classified into three broad categories based on their origin.

- **Ethereum Virtual Machine (EVM):** These honeypots exploit the way the EVM runs its instructions, capitalizing on users' lack of knowledge about the EVM to perform scams such as balance disorder.
- **Solidity Compiler:** These honeypots use the Solidity programming language and manipulate various details about how the code compiles, such as inheritance, variable size, etc. to trick users unfamiliar with Solidity. The techniques used in this category include inheritance disorder, skip empty string literal, type deduction overflow, and uninitialized struct.
- **Etherscan Blockchain Explorer:** These honeypots take advantage of the properties of the Etherscan platform, which is used by a large number of users. The attackers use the platform's features to scam users, such as the fact that internal transactions with zero transaction value are not displayed and that the source code is displayed using an HTML text area with a limited width, which can hide parts of the code. The honeypot techniques used in this category include hidden state update, hidden transfer, and straw man contract.

4.5 Symbolic execution

Symbolic execution is a powerful technique to analyze the behavior of a contract and detect potential vulnerabilities. In this technique, various inputs are assigned to different symbolic variables, and then the program runs with the symbolic values. This generates different execution paths for the program. For each execution path, a list of constraints with symbolic values is generated. A constraint solver is then used to determine the feasibility of each execution path. This method has been widely adopted in smart contract analysis tools such as Mythril

(Sharma & Sharma, 2022), Oyente (Luu *et al.*, 2016), and Securify (Tsankov *et al.*, 2018), among others. However, this approach can face limitations with large programs and complex constraints, which can be mitigated by setting a time limit for the execution.

5. HoneyVader

In this section, we will provide information on the methodology we use to detect honeypots. We present an overview of the HoneyVader's design, our tool for identifying honeypots in Ethereum. Finally, we explain how we expand Honeybadger to detect more honeypots.

5.1 Problem formulation

To better address the issue of honeypot scams in the Ethereum blockchain, we have classified honeypot smart contracts into two distinct categories based on their lifetime: "intrinsic" and "non-intrinsic".

- **Intrinsic Honeypots:** The initial group consists of contracts that are honeypots from the inception of their life cycle, specifically prior to deployment. These contracts do not rely on external interactions to function as honeypots, and their fraudulent nature can be easily identified through testing in a test environment such as Remix IDE.
- **Non-Intrinsic Honeypots:** These contracts are initially benign and do not possess any fraudulent characteristics prior to deployment. However, they can become honeypots after deployment through interactions with other contracts. These interactions trigger specific conditions that transform the contract into a honeypot. Non-Intrinsic Honeypots are thus dependent on external interactions to function as honeypots, making them a unique category within the honeypot landscape.

5.2 Methodology

In this section, we first provide our main idea for detecting intrinsic honeypots starting by defining a new type of contract, called Unique Owner Transfer (UOT) contract which is suspicious of

Table-A I-1 The truth table to detect a honeypot based on the cash flow of a contract

CF-In Anyone	CF-Out (Owner - OwnerControlled)	Cash Flow	Honeypot
false	false	false	false
false	true	false	false
true	false	true	false
true	true	true	?

being a honeypot, and then propose a way to distinguish between malicious UOTs and benign ones. Finally, we present a design overview of HoneyVader.

5.2.1 Unique Owner Transfer

The authors in (Torres *et al.*, 2019) describe the typical behavior of honeypots as follows: the contract creator deploys the contract with an intentional vulnerability, allowing anyone to send funds to the contract with the aim of exploiting the flaw and profiting. The owner of the contract then drains the balance of the contract along with the money sent by the victims. This behavior requires the contract to have the capability to both receive and send funds, referred to as *Cash Flow In (CF-In)* and *Cash Flow Out (CF- Out)*.

Before getting into our heuristic, it is important to explain *owner-controlled* transactions in Solidity. There are two types of transactions in Solidity: restricted and unrestricted. Restricted transactions are designed to only be executed by certain authorized users or contracts, and will reject all other attempts to call them. Restricted transactions can be implemented using various techniques in Solidity. For example, a contract can include a modifier that verifies the identity of the caller, such as a specific address or a contract that has been previously authorized. Another technique is to include a function that requires a specific access level or permission, and only users with that level or permission can call it. Unrestricted transactions on the other hand can be called by anyone on the blockchain network. Based on these definitions, if a transaction that includes a money transfer is restricted to the owner of a contract, we call it owner-controlled cash flow out. Moreover, if a transaction is restricted to be called only by a hard-coded address

in the contract, it is also owner controlled as the hard-coded address is specified by the owner of the contract and users have no control over it.

According to the definitions presented in the first paragraph of this subsection, a contract can only be considered a honeypot if it receives funds from multiple sources but transfers only to the owner or if the transfer is owner controlled. In other cases, where arbitrary users make it to withdraw funds, it is definitely not a honeypot (see in Table I-1). However, if the owner is changeable within the contract, anyone can claim ownership and withdraw the funds and the cash flow out of the contract would not exclusively go to the attacker and the contract would not be a honeypot. As such, the only scenario in which a contract can be suspected of being a honeypot is when all of the cash flow out is directed solely toward an unchangeable owner.

With this in mind, we narrow down the definition of a honeypot to the following criteria:

- The contract must have the ability to receive funds from any arbitrary user.
- The contract must transfer funds exclusively to the owner of the contract.
- The claiming of ownership by users must be impossible.

Table-A I-2 The truth table to detect a UOT contract based on the cash flow

CF-In	CF-Out (Owner - OwnerControlled)	Owner Change	UOT	Honeypot
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	false	false
true	true	false	true	maybe
true	true	true	false	false

We call a contract with the above criteria, a *Unique Owner Transfer (UOT)* contract (see in Table I-1). UOT contracts are the most potent honeypots because:

1. There is no way to withdraw the money out of a contract other than claiming ownership.
2. There is no way to claim ownership of the contract.

This model identifies a contract as a possible honeypot and thus warns the user that 1) there is no way to withdraw the money out of a contract other than claiming the ownership 2) there is no way to claim the ownership of the contract. Having this information, a user is more cautious about interacting with a UOT smart contract that is asking them to invest money. Obviously, not all UOT contracts are malicious and there are benign contracts that take money from any source but only transfer it to the owner of the contract.[5]

5.2.2 Tricks in code

Our research on detected honeypots reveals that the distinction between benign and malicious contracts can often be traced to the presence of deceptive code tricks. These tricks manipulate the user into believing they have the ability to withdraw funds or transfer ownership, even though these operations are impossible in a UOT contract. To further determine the presence of a honeypot, we propose two additional heuristics in the following to be applied to the UOT contracts. This multi-layered approach provides a more robust and thorough method of detecting and avoiding honeypots in the smart contract ecosystem.

- **Fake Money Transfer:**

```
1 contract Lottery {
2     address public owner = msg.sender;
3     ...
4     function guess(uint8 number) public payable {
5         if (keccak256(number) == secretNumberHash && msg.value > this.
              balance) {
6             msg.sender.transfer(this.balance + msg.value);
7         }
8     }
9 }
```

Listing I.1: A simplified example of a fake money transfer

[5] The address of a benign UOT contract: 0xfdc39e06a7297268a0f6d5bd1692ae5fa9026152

In most honeypots, the trap is hidden inside a money transfer misleading the victim to exploit the contract with the hope of getting the money from it. However, this money transfer is fake and either it never executes, or its value is zero despite the victim's expectations. An example is shown in Listing I.1. According to this contract, the user gets paid by guessing the right number and sending some ETH. However, this contract shows a balance disorder honeypot and the money transfer in line 6 will not add any funds to the victim's account.

- **Fake Ownership Change:**

```
1  contract DickMeasurementContest {
2      address owner = msg.sender;
3      modifier onlyowner {
4          require (msg.sender == owner);
5          _;
6      }
7      function mineIsBigger() public payable {
8          if (msg.value > this.balance) {
9              owner = msg.sender;
10         }
11     }
12     function withdraw() public onlyowner {
13         ...
14         msg.sender.transfer(this.balance);
15     }
16     function kill() public onlyowner {
17         if(this.balance == 0)
18             selfdestruct(msg.sender);
19     }
20 }
```

Listing I.2: A simplified example of fake ownership change in a contract

Some attackers delude the victims by placing a fake change of ownership in the code. In this case, the victim attempts to exploit the bogus leak and claim ownership of the contract but the functionality does not actually work as intended. An example of this trick is shown in

Listing I.2 where a naive user might think they can demand ownership of the contract by calling the function `mineIsBigger` and transferring funds to the contract. However, this code snippet is an example of the "balance disorder" honeypot technique (Torres *et al.*, 2019) which states that the balance of the contract increments before the execution of the smart contract and therefore the if statement in line 5 is never going to be true.

5.3 Design overview

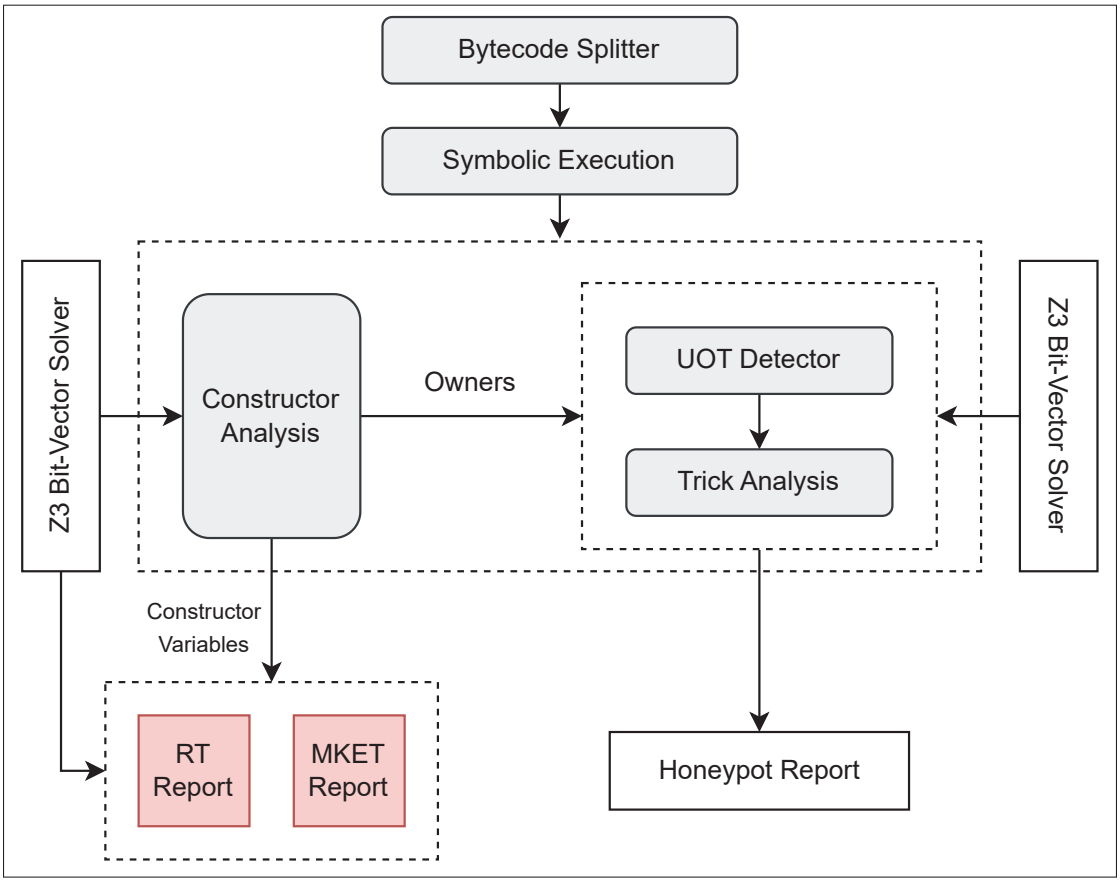

Figure-A I-1 An overview of the HoneyVader's design showing the main components in grey

In this section, we present a design overview of our model. The architecture of our tool, as illustrated in Figure I-1, consists of five primary modules in grey. These modules work in a sequential manner, where each step relies on the information generated from the previous step.

It is important to note that these modules cannot function independently, and each one is an integral component of the overall tool.

The process of detecting a honeypot begins by giving the creation bytecode as the input into the BytecodeSplitter, which separates the constructor bytecode and runtime bytecode. The constructor bytecode is then symbolically executed using the SymbolicExecution module, and the results are analyzed by the ConstructorAnalysis module to extract information about the constructor parameters. Subsequently, the runtime bytecode undergoes symbolic execution and the results are passed on to the UOT Detector and TrickAnalysis. UOT Detector module takes the owner variable as the input and evaluates whether all the funds going out of the contract are directed to the owner, based on the extracted information about the owners and money flows within the contract. If a contract gets labeled as UOT, the TrickAnalysis module is utilized to determine whether the contract is indeed a honeypot. Moreover, we use the constructor variables as the results of constructor analysis and use them to add two new modules to Honeybadger to detect racing time and map key encoding trick honeypots. As it is shown in the design overview, for each component we also use Z3 as the constraint solver.

5.4 Expanding Honeybadger

Honeybadger has a modular design, which allows for the easy expansion and detection of new types of honeypots by incorporating appropriate heuristics into the main module. The constructor variables that are extracted during the constructor analysis process can be utilized to develop new heuristics for detecting various pre-defined honeypot techniques. In addition to the development of a new honeypot detection method, the results of the constructor analysis can be used to develop two new heuristics for detecting racing time and map key encoding trick honeypots introduced in previous works, which can then be integrated into Honeybadger's detection module.

The first heuristic uses the constructor variables to identify instances of racing time honeypots, which involve the use of time-based conditions to trick users into depositing money and

withdrawing more but leaving no time for them to do the withdrawal transaction. By analyzing the path conditions of the contract calls, Honeybadger will be able to determine if a racing time honeypot is in play by checking for variables that are less than or greater than a certain timestamp value. The second heuristic is designed to detect map key encoding trick honeypots, which involve the manipulation of mapping variables to deceive users. By examining the list of hash values associated with the mapping variables and comparing them to the hash values used in the contract, Honeybadger will be able to identify if a map key encoding trick is being employed. With the addition of these two heuristics, Honeybadger's detection capabilities are expanded, providing a more robust and comprehensive solution for detecting honeypots in Solidity contracts.

6. Implementation

This section includes details about the implementation of our tool. We implement our tool in Python with roughly 4,000 lines of code. In the following subsection, we describe the details of each component.

6.1 Splitting bytecode

The BytecodeSplitter module plays a crucial role in our methodology by extracting the bytecode related to the constructor parameters from the EVM creation bytecode. The input to this module can be either the solidity source code or directly the EVM bytecode. In the case of solidity source code, in an additional step, we compile it to the bytecode using `solc` compiler with the command `solc -bin` to get the binary of the creation bytecode.

The creation bytecode is in the form of a hex number, which starts with the constructor bytecode and is followed by the runtime bytecode. Depending on the Solidity version, the constructor bytecode ends with a distinct set of opcodes that form the splitting line. In the compiler version 0.4., the constructor bytecode ends with the opcodes CODECOPY, PUSH1 0X00, RETURN, and STOP. However, for version 0.5 to 0.8, the constructor bytecode ends with CODECOPY, PUSH1 0X00, RETURN, and INVALID.

6.2 Symbolic execution

In our tool, we have adopted and modified the symbolic analysis model introduced by (Luu *et al.*, 2016). This model symbolically executes the bytecode, explores all potential execution paths, and detects the constraints that must be satisfied for the program to behave in a particular manner. Oyente utilizes basic blocks to depict a sequence of instructions in the Ethereum Virtual Machine (EVM) code that are executed one after the other without any branches or jumps in between. For each block, it then evaluates if a path is feasible through the use of a Z3 constraint solver which takes a set of constraints as input and determines if there is any assignment of values to the variables that satisfies all the constraints. A feasible path refers to a specific execution path of a program that adheres to the constraints of the inputs and other specified conditions during the analysis. We define an opcode in a basic block satisfiable if there is a feasible execution path to it.

As a result of the symbolic analysis performed on the constructor bytecode and runtime bytecode, we obtain useful information to guide the next steps. This information is separated for each bytecode and turns into the input for each module.

- **Constructor:**
  - A list of execution paths P.
  - A list of storage writes CON_S made during the SSTORE opcode, which is represented as a tuple $(con\_s_{var}, con\_s_{val})$ where con_s_{var} is the stored variable in the constructor, and con_s_{val} is the stored value in the constructor.
- **Runtime:**
  - A list of execution paths P.
  - A list of storage writes S produced during the SSTORE opcode represented as a tuple (s_{var}, s_{val}) where s_{var} is the stored variable and s_{val} is the stored value.
  - A list of calls C made during the CALL or DELEGATECALL opcode, represented as a tuple (c_r, c_v, c_t, c_{pc}) where c_r is the recipient, c_v is the call value, c_t is the type of call, and c_{pc} is the program counter of the opcode.

- A list of suicides SC made during the SUICIDE or SELFDESTRUCT opcode, represented as a tuple (sc_r, sc_v) where sc_r is the recipient, and sc_v is the suicide value.

6.3 Constructor analysis

Smart contracts are deployed by an owner address, which is a unique public key associated with a wallet. Some contracts store the address of the creator in a constructor variable during the initial deployment transaction and are thus referred to as *owner-aware*. This allows us to retrieve information about the original contract creator. In contrast, other contracts do not store the address of the creator in any variables, making them *non-owner-aware*. This results in a lack of information about the original owner of the contract. After the completion of the symbolic analysis on the constructor bytecode, we can obtain information about the constructor parameters, including a list of owners, O. To identify the owners of the contract, we perform an iteration over all storage writes of the constructor, CON_S, and verify if there exists a storage write, s, where we can prove the equality of s_{val} and I_s using the Z3 solver. If such a storage write exists, the corresponding s_{var} is added to the list of owners, O. Finally, if O is not an empty set after the iteration, it indicates the detection of an owner-aware contract and goes to the next module. Otherwise, the contract is non-owner-aware and it will not continue preceding.

6.4 UOT detector

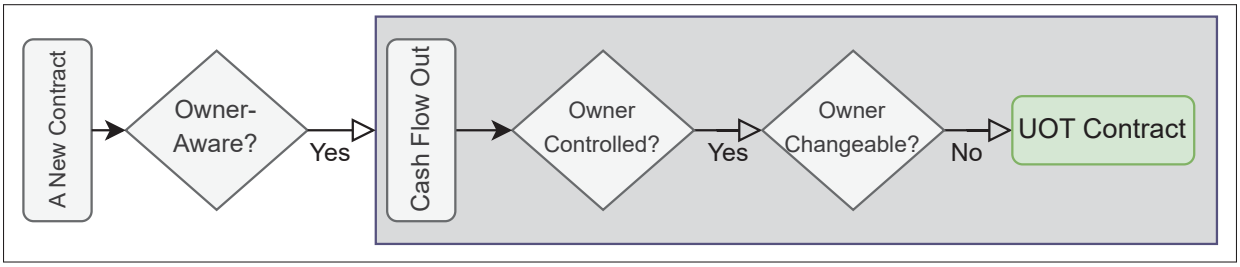

Figure-A I-2 An overview of the workflow of UOT detector module

The overall overview of the UOT detector is shown in Figure 3.4.

To detect a UOT contract, we iterate over all of the satisfiable calls C and suicides SC and check whether, for each call c (or sc), there exists an $owner$ contained in O who controls the cash flow out meaning: 1) we can prove c_r (or sc_r) is equal to the $owner$ using Z3 solver, or 2) c_r is a constant address meaning it is hard coded in the contract, or 3) the path conditions of c (or sc) contains a comparison between I_s and $owner$ meaning only $owner$ can trigger c (or sc), or 4) the path conditions of c contains a comparison between I_s and a constant address meaning a hard coded address by the contract's owner can trigger c (or sc).

If one of these conditions is true for c, we have to check if $owner$ is changeable in the code. To detect this, we iterate over all of the storage writes S and check whether there is a storage write s, where s_{var} is equal to $owner$. If $s \in \emptyset$, $owner$ is unchangeable and as a result of this stage, we add it to the list of funded unchangeable owners FUO. We also add c_{pc} to the list of satisfiable calls to the owner SCO.

After iterating over all of the calls contained in C and suicides contained in SC, we detect CFOFO, if $|SCO| = |C| + |SC|$

6.5 Trick analysis

If the contract is a UOT contract, we proceed to identify two potential tricks in the contract:

- **Fake Money Transfer** We detect a fake money transfer by iterating through the list of calls C and checking if there exists a call c such that $c \notin SCO$, $c_v > 0$, and $c_r \neq owner$ for any $owner$ in O.
- **Fake Ownership Change** To detect any fraudulent ownership changes, we traverse the list of owners O and verify that for each $owner \in O$ there exists corresponding storage write s whose variable, s_{var}, is equal to $owner$, provided $owner \notin FUO$.

6.6 Honeybadger's expansion

As is shown in Figure I-1, we use the results of the constructor analysis to add two new modules to Honeybadger to detect new types of honeypots.

6.6.1 Racing time

To detect racing time in a smart contract, we first iterate over all the function calls contained in the contract and check if there exists a call c, whose path conditions consist of a comparison for the timestamp. We then collect a set of variables less than a specific timestamp value IH_s, and a set of variables greater than IH_s. Next, we examine the storage variables of the constructor and replace the variables corresponding to the less than and greater than sets with their respective values. Finally, we report a racing time if we find a comparison of lt (less than) and gt (greater than) such that $lt < IH\_s < gt$ and the difference between gt and lt is less than 5 minutes. This method enables us to identify potential racing time vulnerabilities in the smart contract

6.6.2 Map key encoding trick

In Solidity, a mapping is a key-value store that is commonly used to store variables. The hash of the map key is used as the address to store the variable in the mapping. However, this method of storing variables can be vulnerable to map key encoding tricks. To detect map key encoding tricks, we first iterate over a list of hashes contained in H. We check if there exists a hash h, whose value is used to store a variable both in the constructor and in the code. We then save the 'raw string' of the hash as a map key used in the contract. We also iterate over all the function calls contained in the contract C. We check if there exists a call c, where the map key m is used in its path conditions, c_r or c_v. We use the map key list to detect any instances of a map key encoding trick. Specifically, we look for a key in the map key list whose UTF-8 encoded form is the same as m, but has a different ASCII representation. If we find such an instance, we can flag it as a map key encoding trick.

7. Evaluation

In this section, we first present the parameters of our experiments and then propose our final results. Then we compare our results to the state-of-the-art tool Honeybadger which also

uses symbolic execution for detecting honeypots. We continue by analyzing one of the newly discovered honeypot techniques and lastly, we present the limitations of our research.

7.1 Experiments

In this section, we present the setup we used for the experiments and the dataset we used. The setup includes details on the hardware and software configurations of the machine used to run the experiments, as well as the parameters used for each experiment. The dataset includes information on the contracts deployed in Ethereum. By providing this information, we aim to provide a clear understanding of the experimental conditions and data used in our study.

Setup: For our experiments, we utilized a computing cluster consisting of two nodes. Each node was equipped with a single Intel Xeon Gold 5118 CPU, which boasts 48 cores operating at a frequency of 2.30GHz. The cluster was running on a 64-bit Linux kernel version 5.15.0-47-generic, which provided a stable and reliable environment for our experiments.

To obtain the initial bytecode of the contracts, we used the Solc compiler version 0.4.25, which is a popular Solidity compiler that compiles smart contracts into EVM bytecode. We then utilized version 1.8.16 of Geth's EVM to retrieve the assembly from the bytecode. This version of Geth's EVM was selected for its stability and compatibility with the version of Solc that we used. The combination of Solc and Geth's EVM provided a reliable and consistent way to generate the necessary bytecode and assembly for our experiments.

The symbolic execution setup included the following configurations:

- Z3 constraint solver version 4.7.1.
- Loop limit of 10.
- Depth limit for DFS of 50.
- Gas limit of 4 million.
- A timeout of 1 second per Z3 request.
- A timeout of 30 minutes to symbolically execute a contract.

Additionally, a timeout of 10 minutes was set to run all of the heuristics for each contract.

Dataset: Our dataset is comprised of the creation bytecode of smart contracts deployed on Ethereum. Since over 98% of the contracts are duplicates, it was more efficient to store only their hashes instead of the whole bytecode. After eliminating duplicates, we removed contracts with similar bytecode hashes, resulting in a set of 375,962 unique contracts. The initial bytecodes of these contracts were then downloaded and stored as our final dataset.

We conducted HoneyVader experiments on 375,962 unique smart contracts. The overall process took an average completion time of 323 seconds per contract. The results showed that 98% of contracts successfully completed the symbolic execution process and almost all of the contracts successfully completed the application of our heuristics in less than 10 minutes. Out of all the contracts evaluated, 121,876 contracts (32%) demonstrated a code coverage greater than 90%.

7.2 Metrics

In this study, we evaluate the effectiveness of our tool using several metrics.

- **TP (True Positives):** This category represents contracts that our methodology and verification confirm to be honeypots.
- **FP (False Positives):** This group includes the contracts that were flagged as honeypots by our tool but are not verified honeypots in the verification phase.
- **FN (False Negatives):** This class of contracts refers to the ones that were detected as benign contracts by mistake but are actually honeypots. We use this metric to compare HoneyVader to Honeybadger.
- **Precision:** P is the proportion of true positives among all predicted positives and we use it to evaluate our detection tool. It is calculated as:

$$p = TP/(TP + FP) \tag{A I-1}$$

7.3 Results

In this section, we present the results of our analysis based on the metrics defined in section 4.2. Our study successfully extracted constructor parameters for all the contracts in the bytecode dataset. Specifically, out of the analyzed contracts, we identified 16,687 that were owner-aware, and these were further analyzed for potential honeypot activity. From this group, we uncovered 4,399 UOT contracts that accepted funding from any source but transferred it to only one address, which was invariably the owner's address. Given that such behavior is indicative of potential honeypot activity, we considered these contracts suspicious and marked them for further investigation. Overall, HoneyVader labeled 139 contracts as honeypots. From this

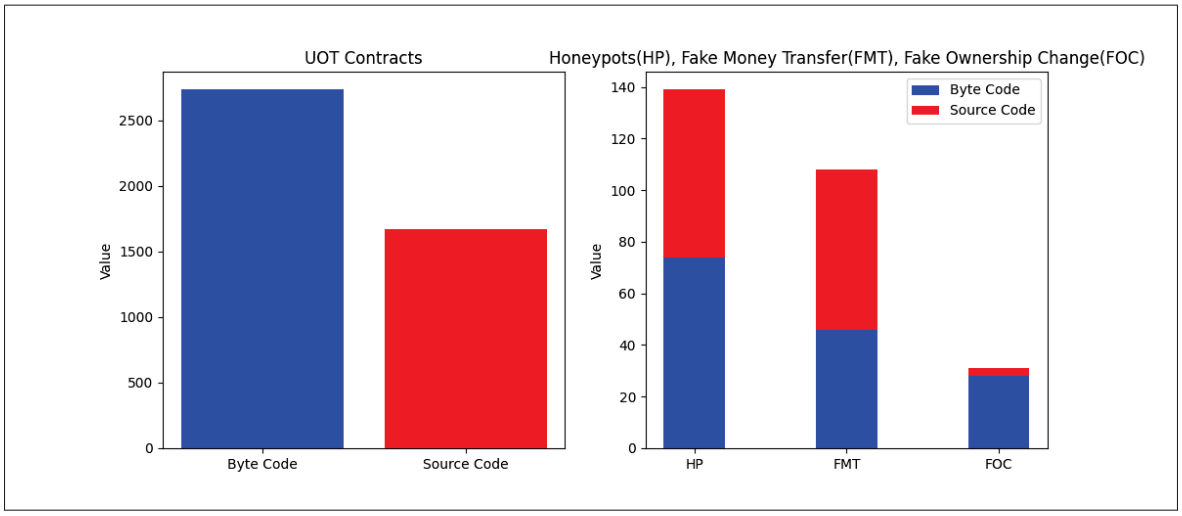

Figure-A I-3 Number of contracts with source code and bytecode for UOT contracts, honeypots using fake money transfer, honeypots using fake ownership change and honeypots overall

number 65 of them are open-source and the rest have only their bytecode available. Our analysis has revealed that 108 of the suspicious contracts employ a fake money transfer method to deceive victims, while 31 entice them into taking ownership of the contract with the hope of withdrawing money. These results emphasize the critical importance of exercising caution when interacting with contracts that exhibit the UOT characteristic. To visually illustrate our primary findings, we have included a figure that clearly displays the frequency of UOT contracts and each type of

attack method used by attackers(see Figure I-3). Interestingly, our results indicate that attackers tend to rely more on the fake money transfer method, as opposed to the fake ownership change, in their honeypot schemes.

7.4 Validation

To validate our results about honeypot contracts, we conducted a thorough manual analysis of the detected honeypots with their source code available. Our validation process consisted of both dynamic and manual analysis.

Based on our results, 65 contracts detected as honeypots have their source code available. For our first step of validation, we used Etherscan (Etherscan, n.d.) to obtain the source code of each one of these contracts and manually analyzed the code for any potential honeypot tricks. To verify our manual analysis, we then ran each contract in Remix IDE (Remix IDE, n.d.) and performed transactions in a test environment to observe the contract's behavior.

In addition, we utilized transaction behaviors as a form of validation for these 65 contracts. Honeypots that were successful in trapping victims show a rug pull pattern in their transactions, where they wait for a victim to deposit funds before pulling all the funds out of the contract. Along with normal transactions that show ETH transfers carried out through a smart contract, we also analyzed internal transactions, which show the outgoing funds of the contract triggered by an external transaction. These transactions reveal the accounts to which the funds have been transferred.

Finally, we conducted a comprehensive review of the comment section on Etherscan, which often includes valuable information and warnings about scams or honeypots. This enabled us to gather additional insight into the potential threats posed by the contracts we analyzed. In particular, we searched for comments that mentioned issues or red flags associated with the contracts, such as suspicious or fraudulent behavior. Additionally, this helped us ensure the accuracy and reliability of our findings, as we were able to confirm the existence of certain contracts and their associated risks, based on the feedback and experiences shared by other users.

Table-A I-3 Number of the evaluation set, owner-aware contracts, UOT contracts, true positives and false positives for each honeypot tricks

Evaluation Set	**Owner Aware**	**UOT**	**Honeypot**			
			Fake Money Transfer		**Fake Ownership Change**	
375,962	16,687	4,399	TP	FP	TP	FP
			57	5	2	1

The verification of our results is summarized in Table I-3. Overall out of 4,399 suspicious UOT contracts, we detect 65 honeypots. From this number, 59 of them are true positives as well as 6 false positives which happened as the result of our limitations explained in subsection 8. Overall, our tool achieves a 90% precision in detecting intrinsic honeypots, regardless of the technique used in them.

7.5 Baseline solution

Table-A I-4 Comparing Honeybadger's results to HoneyVader

	Honeybadger		**HoneyVader**				
	TP	FP	Owner Aware	UOT		Trick Found	
				TP	FP	TP	FP
BD	20	-	20	19	-	19	-
ID	41	7	47	43	1	-	-
SESL	12	-	12	-	-	-	-
TDO	4	-	4	4	-	4	-
US	32	-	31	17	-	16	-
HT	12	-	12	-	-	-	-

In order to evaluate the performance of our honeypot detection tool, we selected Honeybadger as a baseline model. Honeybadger is a widely recognized tool for honeypot detection and employs the same analysis methodology as our tool. By comparing HoneyVader to Honeybadger, we can better understand the strengths and weaknesses of our approach. To conduct the comparison, we performed two tests. The first test involved analyzing the baseline dataset of Honeybadger's results separately for each intrinsic honeypot technique. Since Honeybadger presents its performance results separately for each honeypot technique, we discussed the

comparison for each technique individually. We also summarized our results in Table 4.2, which provides a comprehensive overview of the performance comparison between our tool and the Honeybadger baseline.

- **Balance Disorder:** Our results indicate that our tool successfully identified all 19 UOT contracts in the balance disorder dataset, with the exception of one that could not be detected due to the limitations of the Z3 solver. Additionally, we detected one zero-day honeypot that utilized a balance disorder technique.
- **Inheritance Disorder:** Regarding the inheritance disorder dataset, our tool found 43 UOT contracts among the 48 honeypots detected by Honeybadger. Combining our UOT heuristic with Honeybadger's inheritance disorder heuristic showed an increase of 6% in precision. To make this combination, we first ran the UOT detector on all 48 honeypots flagged by Honeybadger and detected 44 of them as UOT. Then we ran the ID heuristic on the 44 UOT contracts resulting in 44 flagged honeypots. From this number, 4 of them were false positives while 40 of them were detected as true positives. We calculate the precision of Honeybadger and (HoneyVader + Honeybadger) as follows:
  - **Precision of Honeybadger:** 41 / (41 + 7) = 0.85
  - **Precision of (Honeybadger + HoneyVader):** 40 / (40 + 4) = 0.91
- **Skip Empty String Literal - Hidden Transfer:** Unfortunately, due to limitations of our symbolic execution engine in handling `throw` in Solidity, we were unable to detect any UOT contracts in the skip empty string literal - hidden transfer dataset using HoneyVader. However, our tool discovered three new zero-day honeypots with a hidden transfer technique that were not detected by Honeybadger.
- **Uninitialized Struct:** In the uninitialized struct dataset, we identified 17 UOT contracts among 31 owner-aware contracts. After validating the non-UOT contracts, we discovered that 13 of the contracts detected as true positive by Honeybadger were not honeypots, and in five of them, we observed users withdrawing money from the contract. This highlights using UOT heuristic before trying to find honeypots. Moreover, one of the detected honeypots also did not utilize the uninitialized struct technique, highlighting the importance of detection

tools being independent of specific honeypot techniques. Lastly, we found one zero-day honeypot in this dataset.

During the second round of experiments, we applied HoneyVader to all the datasets used by Honeybadger, which included 151,935 unique contracts. Within this dataset, we discovered 7 instances of false negatives that were missed by Honeybadger. Out of these, 5 contracts used techniques previously identified in (Torres *et al.*, 2019), while the remaining 2 employed a new technique. However, the specific technique used is not our primary concern; what matters most is whether the contract is UOT and capable of deceiving the user. Table III-1 contains the addresses of these honeypots.

8. Limitations

Analyzing smart contracts' bytecode is challenging. Despite the advantage of detecting honeypots based on bytecode, the limited availability of open-source contracts (only 2% of all contracts) makes it difficult to identify certain types of honeypots. Some information is lost during the compilation process from source code to bytecode, such as operations that are present in the source code but do not change the state of the contract. For instance, a newly proposed honeypot technique, unexecuted call honeypot (Camino *et al.*, 2020), involves the attacker performing a fund transfer through a call without the necessary parenthesis to execute the call. This results in the call not appearing in the bytecode and only being visible in the source code. Our tool can still detect this contract as UOT, as it is an intrinsic honeypot, but finding the trick in such contracts can be difficult, as the bytecode does not provide any information about the call.

9. Conclusion

In this thesis, we have presented a novel approach to detect honeypot scams in Ethereum smart contracts at the bytecode level through symbolic execution. Our findings demonstrate that analyzing creation bytecode instead of runtime bytecode provides valuable information about constructor parameters, enabling the development of a heuristic to detect honeypots without prior knowledge of the specific trick used. Our approach was able to detect different types of

honeypots based on the Honeybadger classification and identified a new class of honeypots. Moreover, we expand Honeybadger to detect more honeypots. In conclusion, our approach to detecting honeypot scams in Ethereum smart contracts offers a significant contribution to the field and provides a valuable tool for protecting against malicious actors. While our research focused on honeypots in smart contracts, there is also a growing concern around honeypot tokens especially in Binance Smart Chain (BSC), that encourage users to buy the token and then lock their money inside the contract, limiting their access to sell the token or cash out from the contract. We believe that investigating these honeypots in both Ethereum and BSC would be an important direction for future research.

APPENDIX II

SOURCE CODE OF THE IMPLEMENTATION

```
def find_the_owners():
    global owners
    global owner_aware
    owners = []
    owner_aware = False
    if constructor_variables["sstore"]:
        for con_var in constructor_variables["sstore"]:
            if is_expr(con_var["variable"]) and is_expr(con_var["value"]):
                owner_found_by_tx_origin = provee(Extract(159, 0, con_var["
                    value"]) == Extract(159, 0, tx_origin))
                owner_found_by_message_sender = provee(Extract(159, 0,
                    con_var["value"]) == Extract(159, 0, message_sender))

                if owner_found_by_tx_origin == True or
                    owner_found_by_message_sender == True:
                     owners.append(con_var["variable"])
                     owner_aware = True
```

Listing II.1: Simplified code to find the owner of a contract

```
def owners_are_funded():
    global owners
    global sat_calls_to_owner
    global owners_that_are_funded_and_do_not_change
    unchangeable_owners = list(owners)
    changeable_found = True
    while(changeable_found):
        changeable_found = False
        for owner in owners:
            if owner not in unchangeable_owners:
                continue
```

```
12              if ownership_can_be_changed(owner, unchangeable_owners):
13                  unchangeable_owners.remove(owner)
14                  changeable_found = True
15                  break
16      recipients_and_path_conds_and_pc = get_cash_flow_recipients()
17      if not recipients_and_path_conds_and_pc:
18          return None
19      some_owner_is_funded_in_this_call = []
20      for recipient_and_path_cond in recipients_and_path_conds_and_pc:
21          owner_is_funded_and_can_not_be_changed = []
22          recipient = recipient_and_path_cond[0]
23          for owner in owners:
24              owner_funded = False
25              # 1. owner_address = 0x7a617c2B05d2A74Ff9bABC9d81E5225C1e01004b
26              if isinstance(recipient, long) or isinstance(recipient, int):
27                  owner_funded = True
28              # 2. owner.transfer(balance)
29              if not owner_funded:
30                  if is_expr(recipient):
31                      owner_funded = provee(Extract(159, 0, recipient) ==
                            Extract(159, 0, owner), recipient_and_path_cond[1])
32              # 3. require(msg.sender == owner)
33              if not owner_funded:
34                  owner_funded = provee(Extract(159, 0, message_sender) ==
                        Extract(159, 0, owner), recipient_and_path_cond[1])
35              # 4. Extract(159, 0, Is) ==
                    698670862888103124090043688033161627232733560907
36              # require(Is == Real Address)
37              if not owner_funded:
38                  s = Solver()
39                  s.set("timeout", global_params.TIMEOUT)
40                  s.add(recipient_and_path_cond[1])
41                  if is_expr(recipient) and s.check() == sat:
42                      models = s.model()
43                      value = models.eval(recipient)
```

```
                    owner_funded = provee(Extract(159, 0, message_sender)
                        == Extract(159, 0, value), recipient_and_path_cond
                        [1])
            if owner_funded == True:
                owner_changed = ownership_can_be_changed(owner,
                    unchangeable_owners)
            else:
                owner_changed = False
            if owner_funded == True and owner_changed == False:
                owner_is_funded_and_can_not_be_changed.append(True)
                owners_that_are_funded_and_do_not_change.append(owner)
            else:
                owner_is_funded_and_can_not_be_changed.append(False)
        if any(owner_is_funded_and_can_not_be_changed):
            some_owner_is_funded_in_this_call.append(True)
            sat_calls_to_owner.append(recipient_and_path_cond[2])
        else:
            some_owner_is_funded_in_this_call.append(False)
    if all(some_owner_is_funded_in_this_call):
        return True
    return False
```

Listing II.2: Simplified code to detect a UOT contract

```
def ownership_can_be_changed(owner, owners_that_cant_be_changed):
    for sstore in list_of_sstores:
        if not (str(sstore["variable"]) == str(owner)):
            continue
        s = Solver()
        s.set("timeout", global_params.TIMEOUT)
        s.add(sstore["path_condition"])
        satisfiable = s.check()
        if satisfiable == unsat:
            continue
        if isinstance(sstore["value"], (long, int)):
```

```
12              continue
13          if satisfiable == sat:
14              models = s.model()
15              value = models.eval(Extract(159, 0, message_sender))
16              if provee(Extract(159, 0, message_sender) == Extract(159, 0,
                    value), sstore['path_condition']):
17                  continue
18          dependent_on_owner = False
19          for own in owners_that_cant_be_changed:
20              dependent_on_owner = provee(Extract(159, 0, message_sender) ==
                    Extract(159, 0, own), sstore['path_condition'])
21              if dependent_on_owner:
22                  break
23          if dependent_on_owner:
24              continue
25          return True
26      return False
```

Listing II.3: Simplified code to detect if an owner of a contract is changeable in code

```
1  def detect_fake_ownerchange_trick():
2      for owner in owners:
3          for sstore in list_of_sstores:
4              if not is_payable(sstore):
5                  continue
6              if str(sstore['variable']) == str(owner) and (owner in
                   owners_that_are_funded_and_do_not_change):
7              return True
```

Listing II.4: Simplified code to detect a fake ownership change in a UOT contract

```
1  def detect_fake_call_trick():
2      payable_call_trick = False
3      for suicide in list_of_suicides:
4          if sat_calls_to_owner and suicide["pc"] not in sat_calls_to_owner:
```

```
            if not is_payable(suicide):
                continue
            return True
    for index in list_of_calls:
        for call in list_of_calls[index]:
            if call["pc"] in unknown_calls:
                continue
            # either: unsatisfiable call / calling another smart contract /
                sending 0 ethers
            if sat_calls_to_owner and call["pc"] not in sat_calls_to_owner:
                if not is_payable(call):
                    continue
                if is_expr(call["recipient"]) and not str(call["recipient"
                    ]) == "0":
                    calling_another_contract = (not is_expr(call["value"]))
                        and (call["value"] == 0)
                    if not calling_another_contract:
                        print("one unsat or zero money transfer!", call["pc
                            "])
                        payable_call_trick = True
                        call_to_owner = False
                        recipient = get_vars(call["recipient"])
                        for owner in owners:
                            if str(owner) in str(recipient):
                                call_to_owner = True
                                break
                        if (not call_to_owner and payable_call_trick ==
                            True):
                            return True
```

Listing II.5: Simplified code to detect a fake call in a UOT contract

```
def the_map_key_used_in_call(call, hash):
    map_key_address = "Ia_store_" + str(hash['value'])
    for condition in call['path_condition']:
```

```
        if not is_expr(condition):
            break
        if is_in_expr(str(map_key_address), condition):
            return hash['raw_string'].rstrip('\x00')
    if is_expr(call['value']):
        if is_in_expr(str(map_key_address), call['value']):
            return hash['raw_string'].rstrip('\x00')
    if is_expr(call['recipient']):
        if is_in_expr(str(map_key_address), call['recipient']):
            return hash['raw_string'].rstrip('\x00')
    return None
def detect_map_key_encoding():
    all_hashes = []
    all_sstores = []
    map_key_list = []
    for hash in constructor_variables["hash"]:
        all_hashes.append(hash)
    for hash in list_of_hashes:
        all_hashes.append(hash)
    for sstore in constructor_variables["sstore"]:
        all_sstores.append(sstore)
    for sstore in list_of_sstores:
        all_sstores.append(sstore)
    for sstore in all_sstores:
        for hash in all_hashes:
            if hash["value"] == sstore["address"]:
                encoded_string = hash['raw_string'].rstrip('\x00')
                map_key_list.append(encoded_string)
    for index in list_of_calls:
        for call in list_of_calls[index]:
            for hash in all_hashes:
                map_key_in_call = the_map_key_used_in_call(call, hash)
                if not map_key_in_call:
                    continue
```

```
38                  map_key_in_call = unicode(map_key_in_call, "utf-8", errors=
                        'replace')
39                  decoded_map_key_in_call = unidecode(map_key_in_call)
40                  for map_key in map_key_list:
41                      map_key = unicode(map_key, "utf-8", errors='replace')
42                      decoded_map_key = unidecode(map_key)
43                      if map_key != map_key_in_call and decoded_map_key ==
                            decoded_map_key_in_call:
44                          heuristic = {}
45                          heuristic["function_signature"] = call["
                                function_signature"]
46                          heuristic["block"] = call["function_signature"]
47                          heuristic["type"] = HeuristicTypes.MAP_KEY_ENCODING
48                          heuristic["pc"] = call["function_signature"]
49                          if not heuristic in heuristics:
50                              heuristics.append(heuristic)
```

Listing II.6: Simplified code to detect map key encoding trick honeypot

```
1  def detect_racing_time():
2  # the minimum amount of time to do a tx in ethereum is considered to be 5
       mins
3      for index in list_of_calls:
4          for call in list_of_calls[index]:
5              if owner_aware and sat_calls_to_owner and
                   owners_that_are_funded_and_do_not_change:
6                  if call["pc"] in sat_calls_to_owner and is_expr(call["
                       recipient"]):
7                      for owner in owners_that_are_funded_and_do_not_change:
8                          unchangeable_owner = provee(Extract(159, 0, call["
                               recipient"]) == Extract(159, 0, owner))
9                          if unchangeable_owner:
10                             continue
11             condition_args = []
12             comp_args_lt = []
```

```
13              comp_args_gt = []
14              rct_possible = False
15              for condition in call["path_condition"]:
16                  if is_expr(condition):
17                      if "IH_s" in str(condition) and condition in
                            list_of_comparisons:
18                          for con_var in constructor_variables["sstore"]:
19                              for var in get_vars(condition):
20                                  if str(var) == str(con_var['variable']):
21                                      if not is_expr(con_var['value']):
22                                          condition = substitute(condition, (
                                                var, BitVecVal(con_var['value'
                                                ], 256)))
23                                      else:
24                                          condition = substitute(condition, (
                                                var, con_var['value']))
25                          simplified_condition = simplify(condition)
26                          expr_args = get_ULE_expr_args(simplified_condition)
27                          condition_args.append(expr_args)
28              while condition_args:
29                  args = condition_args.pop()
30                  if args:
31                      args = args.pop()
32                      if is_in_expr('IH_s', args[0]):
33                          comp_args_gt.append(args[1])
34                      else:
35                          comp_args_lt.append(args[0])
36              minimum = bv_min(comp_args_gt)
37              maximum = bv_max(comp_args_lt)
38                  if provee(ULE((minimum - maximum), 300), minimum > maximum,
                        *call["path_condition"]) == True:
39                      heuristic = {}
40                      heuristic["function_signature"] = call["
                            function_signature"]
41                      heuristic["block"] = call["block"]
```

```
42                      heuristic["type"] = HeuristicTypes.RACING_TIME
43                      heuristic["pc"] = call["pc"]
44                      if not heuristic in heuristics:
45                          heuristics.
```

Listing II.7: Simplified code to detect racing time honeypot

APPENDIX III

TABLES IN ANNEXES

Table-A III-1 Addresses of zero-day Honeypots

Contract Address
0x08fc24c9128591b74191fcff6833E746BA67d63e
0x26B5962250B779ab0F33970738A46FcFb00a70b9
0x2BB5B9f83391d4190f8b283Be0170570953c5a8e
0x31FD65340A3d272E21Fd6ac995f305CC1AD5f42A
0x5ABb8dDA439BECBd9585D1894Bd96Fd702400fA2
0xB11E0464CBCf0A06BBE29d8882c9FC7a96CcA9FB
0x223b0ee581719d4c6ae36f1ba1dd4101e5409c1c
0x0Dc11B7Ed751594906BCe3a7091952b30528ee7E
0xE952CBf5C1480032537c57874DE3e949203Ed623

BIBLIOGRAPHY

Abdelhamid, N., Ayesh, A. & Thabtah, F. (2014). Phishing detection based Associative Classification data mining. 41(13), 5948–5959. doi: 10.1016/j.eswa.2014.03.019.

Alharby, M. & van Moorsel, A. (2017-08-26). Blockchain-based Smart Contracts: A Systematic Mapping Study. *Computer Science & Information Technology (CS & IT)*, pp. 125–140. doi: 10.5121/csit.2017.71011.

Allen, F. E. (1970a). Control Flow Analysis. *SIGPLAN Not.*, 5(7), 1–19. doi: 10.1145/390013.808479.

Allen, F. E. (1970b). Control Flow Analysis. *Proceedings of a Symposium on Compiler Optimization*, pp. 1–19. doi: 10.1145/800028.808479.

Antonucci, F., Figorilli, S., Costa, C., Pallottino, F., Raso, L. & Menesatti, P. (2019). A review on blockchain applications in the agri-food sector. *Journal of the Science of Food and Agriculture*, 99(14), 6129-6138. doi: https://doi.org/10.1002/jsfa.9912.

Aponte-Novoa, F. A., Orozco, A. L. S., Villanueva-Polanco, R. & Wightman, P. (2021). The 51% Attack on Blockchains: A Mining Behavior Study. *IEEE Access*, 9, 140549-140564. doi: 10.1109/ACCESS.2021.3119291.

Atlam, H., Alenezi, A., Alassafi, M. & Wills, G. (2018). Blockchain with Internet of Things: Benefits, Challenges and Future Directions. *International Journal of Intelligent Systems and Applications*, 10, 40–48. doi: 10.5815/ijisa.2018.06.05.

Atzei, N., Bartoletti, M. & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts SoK. pp. 164–186. doi: 10.1007/978-3-662-54455-6_8.

Badawi, E. & Jourdan, G.-V. (2020). Cryptocurrencies Emerging Threats and Defensive Mechanisms: A Systematic Literature Review. *IEEE Access*, 8, 200021-200037. doi: 10.1109/ACCESS.2020.3034816.

Bai, Q., Zhou, X., Wang, X., Xu, Y., Wang, X. & Kong, Q. (2019, 05). A Deep Dive Into Blockchain Selfish Mining. pp. 1-6. doi: 10.1109/ICC.2019.8761240.

Ball, T. (1999). The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6), 216–234.

Bartoletti, M., Pes, B. & Serusi, S. (2018-06). Data Mining for Detecting Bitcoin Ponzi Schemes. *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 75–84. doi: 10.1109/CVCBT.2018.00014.

Bartoletti, M., Carta, S., Cimoli, T. & Saia, R. (2020-01). Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. 102, 259–277. doi: 10.1016/j.future.2019.08.014.

Ben Ayed, A. (2017-05-30). A Conceptual Secure Blockchain Based Electronic Voting System. 9(3), 01–09. doi: 10.5121/ijnsa.2017.9301.

Beniiche, A. (2020). A Study of Blockchain Oracles. *ArXiv*, abs/2004.07140, 1 – 9.

Bian, L., Zhang, L., Zhao, K., Wang, H. & Gong, S. (2021). Image-Based Scam Detection Method Using an Attention Capsule Network. 9, 33654–33665. doi: 10.1109/ACCESS.2021.3059806.

Bjesse, P. (2005). What is Formal Verification? *SIGDA Newsl.*, 35(24), 1–es. Retrieved from: https://doi.org/10.1145/1113792.1113794.

Bocek, T., Rodrigues, B. B., Strasser, T. & Stiller, B. (2017-05). Blockchains everywhere - a use-case of blockchains in the pharma supply-chain. *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 772–777. doi: 10.23919/INM.2017.7987376.

Brighente, A., Conti, M. & Kumar, S. (2022). Extorsionware: Exploiting Smart Contract Vulnerabilities for Fun and Profit. *ArXiv*, abs/2203.09843, 1–6.

Bryatov, S. & Borodinov, A. (2019-01-01). Blockchain technology in the pharmaceutical supply chain: researching a business model based on Hyperledger Fabric. 134–140. doi: 10.18287/1613-0073-2019-2416-134-140.

Buterin, V. (2016). CRITICAL UPDATE Re: DAO Vulnerability. Retrieved on 2023-02-20 from: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability.

Buterin, V. et al. (2013). Ethereum white paper. 1–36.

Camino, R., Torres, C. F., Baden, M. & State, R. (2020). A Data Science Approach for Detecting Honeypots in Ethereum. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1-9. doi: 10.1109/ICBC48266.2020.9169396.

Cao, X., Zhang, J., Wu, X. & Liu, B. (2022). A survey on security in consensus and smart contracts. *Peer-to-Peer Networking and Applications*, 15, 1-21. doi: 10.1007/s12083-021-01268-2.

Chen, H., Pendleton, M., Njilla, L. & Xu, S. (2020a). A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.*, 53(3), 1–43. doi: 10.1145/3391195.

Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M. H. & Zhang, X. (2017, 12). An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks. doi: 10.1007/978-3-319-72359-4_1.

Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P. & Zhou, Y. (2018). Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, pp. 1409–1418. doi: 10.1145/3178876.3186046.

Chen, W., Zheng, Z., Ngai, E. C.-H., Zheng, P. & Zhou, Y. (2019). Exploiting Blockchain Data to Detect Smart Ponzi Schemes on Ethereum. *IEEE Access*, 7, 37575-37586. doi: 10.1109/ACCESS.2019.2905769.

Chen, W., Guo, X., Chen, Z., Zheng, Z., Lu, Y. & Li, Y. (2020b). Honeypot Contract Risk Warning on Ethereum Smart Contracts. *2020 IEEE International Conference on Joint Cloud Computing*, pp. 1-8. doi: 10.1109/JCC49151.2020.00009.

Chen, W., Guo, X., Chen, Z., Zheng, Z. & Lu, Y. (2020-07). Phishing Scam Detection on Ethereum: Towards Financial Security for Blockchain Ecosystem. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 4506–4512. doi: 10.24963/ijcai.2020/621.

Chen, W., Li, X., Sui, Y., He, N., Wang, H., Wu, L. & Luo, X. (2021). SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(2), 1–30. doi: 10.1145/3460093.

contributors, T. W. (2022). Web3.py: A Python library for interacting with Ethereum. Retrieved from: https://github.com/ethereum/web3.py.

Ellis, E., Luu, L., Chu, D.-H., Wang, H. & Sai, I. (2017). ChainLink: A Decentralized Oracle Network. *Proceedings of the 2017 International Conference on Blockchain*, pp. 7:1–7:7. doi: 10.1145/3139904.3139910.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27.

Ethereum Foundation. (2021). Vyper: the Pythonic Programming Language for Ethereum. Retrieved from: https://vyper.readthedocs.io.

Etherscan. [Accessed: 2022]. (n.d.). Etherscan - Block Explorer, Charts, Transactions & Dapps. Retrieved from: https://etherscan.io/.

Eyal, I. & Sirer, E. G. (2018). Majority is Not Enough: Bitcoin Mining is Vulnerable. *Commun. ACM*, 61(7), 95–102. doi: 10.1145/3212998.

Feist, J., Greico, G. & Groce, A. (2019). Slither: A Static Analysis Framework for Smart Contracts. *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, (WETSEB '19), 8–15. doi: 10.1109/WETSEB.2019.00008.

Finley, K. (2016). A $50 Million Hack Just Showed That the DAO Was All Too Human | WIRED. Retrieved on 2023-02-20 from: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/.

Grech, N., Brent, L., Scholz, B. & Smaragdakis, Y. (2019). Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1176-1186. doi: 10.1109/ICSE.2019.00120.

Grech, N., Lagouvardos, S., Tsatiris, I. & Smaragdakis, Y. (2022). Elipmoc: advanced decompilation of Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6, 1-27. doi: 10.1145/3527321.

Grieco, G., Song, W., Cygan, A., Feist, J. & Groce, A. (2020). Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. (ISSTA 2020), 557–560. doi: 10.1145/3395363.3404366.

Hara, K., Takahashi, T., Ishimaki, M. & Omote, K. (2021). Machine-learning Approach using Solidity Bytecode for Smart-contract Honeypot Detection in the Ethereum. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 652-659. doi: 10.1109/QRS-C55045.2021.00099.

Heilman, E., Kendler, A., Zohar, A. & Goldberg, S. (2015). Eclipse attacks on bitcoin's peer-to-peer network. *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 129–144.

Hu, H., Bai, Q. & Xu, Y. (2022). SCSGuard: Deep Scam Detection for Ethereum Smart Contracts. *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1-6. doi: 10.1109/INFOCOMWKSHPS54753.2022.9798296.

Huang, J., Han, S., You, W., Shi, W., Liang, B., Wu, J. & Wu, Y. (2021). Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Transactions on Information Forensics and Security*, 16, 2144-2156. doi: 10.1109/TIFS.2021.3050051.

Huang, Y., Zhang, T., Fang, S. & Tan, Y. (2022). SmartIntentNN: Towards Smart Contract Intent Detection. *arXiv preprint arXiv:2211.13670*, arXiv–2211.

Jafar, U., Aziz, M. J. A. & Shukur, Z. (2021). Blockchain for Electronic Voting System—Review and Open Research Challenges. *Sensors*, 21(17), 5874. doi: 10.3390/s21175874.

Jain, A., Duin, R. & Mao, J. (2000). Statistical pattern recognition: a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1), 4-37. doi: 10.1109/34.824819.

Jiang, B., Liu, Y. & Chan, W. K. (2018). ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (ASE '18), 259–269. doi: 10.1145/3238147.3238177.

Juels, A., Kosba, A. & Shi, E. (2016). The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (CCS '16), 283–295. doi: 10.1145/2976749.2978362.

King, J. C. (1976). Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), 385–394. doi: 10.1145/360248.360252.

Kosba, A., Miller, A., Shi, E., Wen, Z. & Papamanthou, C. (2016). Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 839-858. doi: 10.1109/SP.2016.55.

Koscina, M., Manset, D., Negri-Ribalta, C. & Perez, O. (2019). Enabling Trust in Healthcare Data Exchange with a Federated Blockchain-Based Architecture. *IEEE/WIC/ACM International Conference on Web Intelligence - Companion Volume*, (WI '19 Companion), 231–237. doi: 10.1145/3358695.3360897.

Krupp, J. & Rossow, C. (2018, aug). teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1317–1333. Retrieved from: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

Kuo, T.-T., Kim, H.-E. & Ohno-Machado, L. (2017). Blockchain distributed ledger technologies for biomedical and health care applications. 24(6), 1211–1220. doi: 10.1093/jamia/ocx068.

Kushwaha, S., Joshi, S., Singh, D., Kaur, M. & Lee, H.-N. (2022a). Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access*, PP, 1-1. doi: 10.1109/ACCESS.2021.3140091.

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. & Lee, H.-N. (2022b). Ethereum Smart Contract Analysis Tools: A Systematic Review. 10, 57037–57062. doi: 10.1109/ACCESS.2022.3169902.

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. & Lee, H.-N. (2022c). Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. 10, 6605–6621. doi: 10.1109/ACCESS.2021.3140091.

Li, H., Zhu, L., Shen, M., Gao, F., Tao, X. & Liu, S. (2018). Blockchain-Based Data Preservation System for Medical Data. 42(8), 141. doi: 10.1007/s10916-018-0997-3.

Li, W., Bu, J., Li, X. & Chen, X. (2022). Security Analysis of DeFi: Vulnerabilities, Attacks and Advances. *2022 IEEE International Conference on Blockchain (Blockchain)*, pp. 488–493. doi: 10.1109/Blockchain55522.2022.00075.

LucaPennella. (2023). Proof-of-stake (PoS). Retrieved from: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P. & Hobor, A. (2016). Making Smart Contracts Smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (CCS '16), 254–269. doi: 10.1145/2976749.2978309.

Luu, L., Velner, Y., Teutsch, J. & Saxena, P. (2017). SMART POOL: Practical Decentralized Pooled Mining. *USENIX Security Symposium*, pp. 1409–1426.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T. & Dinaburg, A. (2019, 11). Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. pp. 1186-1189. doi: 10.1109/ASE.2019.00133.

Nakamoto, S. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System. 1–11. Retrieved from: http://www.bitcoin.org/bitcoin.pdf.

Okupski, T. (2018). Ethereum Virtual Machine (EVM) Illustrated. Retrieved from: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

Perez, D. & Livshits, B. (2021). Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1325–1341. Retrieved from: https://www.usenix.org/conference/usenixsecurity21/presentation/perez.

Petrov, S. (2017). Another Parity Wallet hack explained. Retrieved on 2023-02-20 from: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

Remix IDE. [Accessed: 2022]. (n.d.). Remix - Solidity IDE. Retrieved from: https://remix.ethereum.org/.

Samreen, N. F. & Alalfi, M. H. (2020). A Survey of Security Vulnerabilities in Ethereum Smart Contracts. *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, (CASCON '20), 73–82.

Serrano, A. & Clack, C. (2018). Bamboo: a simple, static and deterministic language for smart contracts. Retrieved from: https://github.com/pirapira/bamboo.

Sharma, N. & Sharma, S. (2022). A Survey of Mythril, A Smart Contract Security Analysis Tool for EVM Bytecode. 13, 51003-51010.

Siegel, D. [Section: Learn]. (2023). Understanding The DAO Attack. Retrieved on 2023-02-20 from: https://www.coindesk.com/learn/understanding-the-dao-attack/.

Solidity. (2016). Solidity Documentation. Retrieved from: https://solidity.readthedocs.io/en/v0.8.9/.

Specific-Economist70. (2022). Is this a honeypot? Retrieved from: https://www.reddit.com/r/solidity/comments/shgd4j/is_this_a_honeypot/.

Su, L., Shen, X., Du, X., Liao, X., Wang, X., Xing, L. & Liu, B. (2021). Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications. *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1307–1324. Retrieved from: https://www.usenix.org/conference/usenixsecurity21/presentation/su.

Takanen, A., Demott, J. D., Miller, C. & Kettunen, A. (2018). *Fuzzing for software security testing and quality assurance*. Artech House.

Tama, B. A., Kweka, B. J., Park, Y. & Rhee, K.-H. (2017a). A critical review of blockchain and its current applications. *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, pp. 109-113. doi: 10.1109/ICECOS.2017.8167115.

Tama, B. A., Kweka, B. J., Park, Y. & Rhee, K.-H. (2017b). A critical review of blockchain and its current applications. *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, pp. 109-113. doi: 10.1109/ICECOS.2017.8167115.

Tama, B. A., Kweka, B. J., Park, Y. & Rhee, K.-H. (2017c). A critical review of blockchain and its current applications. *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, pp. 109-113. doi: 10.1109/ICECOS.2017.8167115.

thec00n. (2018). smart-contract-honey-pots. Retrieved on 2023-02-21 from: https://github.com/thec00n/smart-contract-honeypots.

Tian, H., He, J. & Ding, Y. (2019). Medical Data Management on Blockchain with Privacy. 43(2), 26. doi: 10.1007/s10916-018-1144-x.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E. & Alexandrov, Y. (2018). SmartCheck: Static Analysis of Ethereum Smart Contracts. *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 9-16.

Torres, C. F., Steichen, M. & State, R. (2019). The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1591–1607. Retrieved from: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira.

Torres, C. F., Iannillo, A. K., Gervais, A. & State, R. (2021). ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 103-119.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F. & Vechev, M. (2018). Securify: Practical Security Analysis of Smart Contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, (CCS '18), 67–82. doi: 10.1145/3243734.3243780.

Vasek, M. & Moore, T. W. (2018). Analyzing the Bitcoin Ponzi Scheme Ecosystem. *Financial Cryptography Workshops*, pp. 101–112.

Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R. & Wang, F.-Y. (2018). An Overview of Smart Contract: Architecture, Applications, and Future Trends. *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 108-113. doi: 10.1109/IVS.2018.8500488.

Wang, Z., Jin, H., Dai, W., Choo, K.-K. R. & Zou, D. (2021). Ethereum smart contract security research: survey and future research opportunities. 15(2), 152802. doi: 10.1007/s11704-020-9284-9.

Wen, H., Fang, J., Wu, J. & Zheng, Z. (2021, 05). Transaction-Based Hidden Strategies against General Phishing Detection Framework on Ethereum. *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. doi: 10.1109/ISCAS51556.2021.9401091.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 1–32.

Wu, J., Yuan, Q., Lin, D., You, W., Chen, W., Chen, C. & Zheng, Z. (2022). Who Are the Phishers? Phishing Scam Detection on Ethereum via Network Embedding. 52(2), 1156–1166. doi: 10.1109/TSMC.2020.3016821.

Zhang, F., Cecchetti, E., Croman, K., Juels, A. & Shi, E. (2016, 10). Town Crier: An Authenticated Data Feed for Smart Contracts. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 270–282. doi: 10.1145/2976749.2978326.

Zhang, Q., Wang, Y., Li, J. & Ma, S. (2020, 02). EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 116–126. doi: 10.1109/SANER48275.2020.9054822.

Zhang, W., Banescu, S., Passos, L., Stewart, S. & Ganesh, V. (2019, 10). MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract. *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 456–462. doi: 10.1109/ISSRE.2019.00052.

Zhang, Y., Kang, S., Dai, W., Chen, S. & Zhu, J. (2021, 09). Code Will Speak: Early detection of Ponzi Smart Contracts on Ethereum. *2021 IEEE International Conference on Services Computing (SCC)*, pp. 301–308. doi: 10.1109/SCC53864.2021.00043.

Zhao, Z. & Chan, T.-H. H. (2016). How to Vote Privately Using Bitcoin. In Qing, S., Okamoto, E., Kim, K. & Liu, D. (Eds.), *Information and Communications Security* (vol. 9543, pp. 82–96). Springer International Publishing. doi: 10.1007/978-3-319-29814-6_8.

Zhou, H., Milani Fard, A. & Makanju, A. (2022). The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. 2(2), 358–378. doi: 10.3390/jcp2020019.

Zhou, S., Yang, Z., Xiang, J., Cao, Y., Yang, Z. & Zhang, Y. (2020, aug). An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem. *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2793–2810. Retrieved from: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-shunfan.

Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A. & Bailey, M. (2018). Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1371–1385. Retrieved from: https://www.usenix.org/conference/usenixsecurity18/presentation/zhou.

Zyskind, G., Nathan, O. & Pentland, A. S. (2015, 05). Decentralizing Privacy: Using Blockchain to Protect Personal Data. *2015 IEEE Security and Privacy Workshops*, pp. 180–184. doi: 10.1109/SPW.2015.27.