

國立清華大學
資訊工程學系

碩士論文

以模擬交易來偵測基於自動化造市機制的價格
預言機弱點方法

**A Method of Using Transaction Simulation to
Detect AMM-based Price Oracle Vulnerabilities**

指導教授：黃慶育 博士 (Dr. Chin-Yu Huang)

學 生：黃柏維 (Po-Wei Huang)

學 號：110062659

中華民國一百一十二年七月

Abstract

With the swift evolution and expansion of Distributed Ledger Technology (DLT) and the cryptocurrency market, an assortment of applications built on blockchain has thrived. Nonetheless, the lucrative target also caught the attention of malicious actors, making 2022 the year that has witnessed the highest frequency and total loss in Decentralized Finance (DeFi) attacks. Numerous developers, eager for a quick market entry, overlook fundamental software testing and quality assurance, leading to unexpected consequences or susceptibilities in smart contracts.

Unlike traditional software, smart contracts – once deployed on the blockchain – cannot be patched, and therefore it can underscore the vital importance of testing during development. Automated testing may have restrictions such as missing bugs due to predetermined vulnerability patterns or generating numerous false positives. As blockchain applications evolve, smart contract project architectures and their business logics become increasingly complex, making vulnerabilities or business logic flows difficult to discover. Despite the availability of various analytical tools, to our knowledge, no tools can detect vulnerabilities associated with price oracle manipulation. AMM-based price oracles use liquidity pools as its data feed. Compared to centralized price oracles, such as Chainlink, AMM-based ones are used more widely and are not limited to mainstream cryptocurrencies. However, this also makes them more

vulnerable to attacks due to potential weaknesses.

In this study, we proposed a price oracle manipulation vulnerability detection method by using smart contract message calls with mutated state variables to simulate a price manipulation attack. We design different simulation scenarios in our experiment based on past attack vectors used in DeFi projects. Then the mutated results received from the simulation are compared with the original values to determine whether there might be potential vulnerabilities. We manually review price oracles that are flagged as vulnerable by our method to prevent false alarms. Our method has aimed to provide users, auditors, and developers a fast and effective way to discern, without needing prior knowledge or the project's architecture or the absence of adequate documentation.



Keywords: Software Engineering, Smart Contract, Blockchain, Price Oracle

中文摘要

隨著分佈式賬本技術（DLT）和加密貨幣市場的快速演進和擴展，基於區塊鏈的各種應用得以繁榮發展。然而，這也同樣引起了黑客的興趣，使 2022 年成為目睹去中心化金融（DeFi）攻擊頻率和總損失最高的一年。許多開發人員因為急於快速進入市場，忽視了基本的軟體測試和品質保證，導致智慧合約出現嚴重錯誤或成為容易受攻擊的目標，讓使用者蒙受損失。

與傳統軟體不同的是，一旦在區塊鏈上部署智慧合約，就無法再做修改，這更凸顯了在開發過程中進行測試的重要性。自動化測試工具是有所限制，如只能從預先設計好的模式而檢查智慧合約弱點，或者是產生大量的誤報。而隨著區塊鏈應用的發展，智慧合約項目的架構和商業邏輯也變得越來越複雜，使其弱點或邏輯瑕疵更難以被發現。儘管在市面上還是有各種分析工具可供使用，但據我們所知，沒有工具能夠有效的檢測與價格預言機操縱相關的弱點。以使用自動化造市流動性池（AMM）的價格預言機為其數據源，與如 Chainlink 等中心化的價格預言機相比，基於自動化造市流動性池的預言機使用更為廣泛，且不受限於僅支援主流幣種。然而，這也因為是無監管的方式使它們更易受到攻擊，也因次對於使用項目而言是不可忽略的風險。

在這項研究中，我們提出了一種透過覆寫儲存在智慧合約上數值的智慧合約消息調用來模擬操縱攻擊的價格預言機操縱漏洞檢測方法。我們根據過去對去中心化金融項目的攻擊方式在實驗中設計不同的模擬情境，然後將模擬得到的變異

結果與原始值進行比較，以判定是否可能存在潛在漏洞。我們對被我們的方法標記為可能存在漏洞的價格預言機進行人工檢查，以排除誤報。我們的目標是為用戶、審計人員、開發人員提供一種快速有效的方式，以便在不需事先知道項目的架構或在沒有足夠說明文件的情況下，提前識別出可能存在的價格預言機漏洞。

關鍵字：軟體工程、智慧合約、區塊鏈、價格預言機



Acknowledgement

在清華大學就讀碩士的兩年中，非常感謝身邊的人給我的支持與鼓勵。首先要感謝我的指導教授黃慶育教授，指導我完成這篇跨足不同領域的研究論文，給予了我需多進行研究實驗和撰寫論文的建議，以及軟體工程的見解與經驗。除此之外，也感謝老師對我們在生活、待人處事上的關心與提醒。

我還要感謝軟體工程實驗室的其他人，謝謝志強學長、傳旻學長和宗毅學長對我的關心與照顧；謝謝實驗室的夥伴湘梅與煥澤，能夠一起分擔實驗室的事務以及軟體工程助教的事務，讓我能夠專心自己的研究；還有謝謝實驗室的學弟妹靖蓉、述宇、建賢與沛語，能夠在枯燥無味的生活中有你們的陪伴。

最後，我要感謝我的家人，在求學的過程中一直支持我，給予我信心，讓我能無後顧之憂地完成碩士學位。在清大的這段日子裡，非常感謝你們的陪伴。



Contents

Abstract.....	i
Abstract in Chinese.....	iii
Acknowledge.....	v
List of Figures.....	viii
List of Tables.....	ix
List of Acronyms.....	x
List of Notation.....	xi
Chapter 1 Introduction.....	1
Chapter 2 Background and Related Works.....	7
2.1 Smart Contracts.....	7
2.2 Decentralize Finance.....	7
2.3 Price Oracle Manipulation.....	13
2.3.1 Preventions from Price Oracle Attack.....	17
2.4 ProMutator.....	18
Chapter 3 Methodology.....	20
3.1 Foundry.....	22
3.2 Simulate Function Calls.....	26
3.2.1 Warp Finance Incident.....	31
3.2.2 Inverse Finance Incidents.....	34
3.2.3 Cheese Bank Finance Incidents.....	36
3.3 Generating Mutate Value.....	37
3.4 Prune function calls.....	39
3.5 Detection Rules.....	47
Chapter 4 Experiments and Result.....	49
4.1 Dataset Description.....	49
4.2 Baseline Methods.....	54
4.3 Comparison Criteria.....	55
4.4 Experimental Results.....	56
4.4.1 Output of Transaction Simulation.....	56
4.4.2 Results of Detections and Manual Inspections.....	57
Chapter 5 Threads to Validity and Research Questions.....	64
5.1 Threat to Validity.....	64

5.1.1	Internal Validity.....	64
5.1.2	External Validity.....	65
5.1.3	Construct Validity.....	65
5.1.4	Conclusion Validity.....	66
5.2	Research Questions.....	66
Chapter 6	Conclusion and Future Work.....	69
References.....		71



List of Figures

Figure 1	Graph of CPMM Model.....	9
Figure 2	Market Manipulation on CEX.....	17
Figure 3	Price Oracle Function Call Flow in a Transaction.....	19
Figure 4	Flowchart of Proposed Method.....	22
Figure 5	Foundry Test Example.....	24
Figure 6	Execution Traces from Foundry.....	24
Figure 7	Execution Tree of Transaction.....	28
Figure 8	Example of eth_createAccessList().....	30
Figure 9	Flowchart of Simulation Function Calls.....	31
Figure 10	Slither Result on Warp Finance Oracle.....	33
Figure 11	Comparison of Attack Transaction Invocation Flow and Test Result...33	
Figure 12	Simulation of Manipulating TWAP Price Oracle in Inverse Finance...35	
Figure 13	Possible State of an UniswapV2 Pool Reserves.....	38
Figure 14	Token Flow of a Multi-hop Swap Transaction.....	41
Figure 15	Market Share Pie Chart of Oracles based on TVS.....	51
Figure 16	False Positive of Detecting Price Oracle Vulnerability.....	58
Figure 17	Harmless Price Oracle Vulnerability.....	59
Figure 18	Occurrences Chart of Detected Potential Price Oracle.....	63

List of Tables

Table 1	Summary of Calculating Storage Slot of Different Data Types.....	39
Table 2	Definitions of DeFi Actions.....	44
Table 3	AMM-based Price-related Functions.....	51
Table 4	Dataset Description at Different Timestamp.....	52
Table 5	Dataset Description Compared to ProMutator.....	52
Table 6	Dataset Description of Recent Transactions.....	53
Table 7	Attacks Related to Price Oracle on Ethereum Mainnet.....	54
Table 8	Simulation Results at Different Timestamp.....	60
Table 9	Simulation Results Compared to ProMutator.....	61
Table 10	Simulation Results of Recent Transactions.....	62



List of Acronyms

DLT	Distributed Ledger Technology
DeFi	Decentralized Finance
NFT	Non-Fungible Token
EOA	External Owned Account
DAO	Decentralized Autonomous Organization
CPMM	Constant Product Market Maker
TWAP	Time Weighted Average Price
TWAT	Time Weighted Average Tick
UI	User Interface
EVM	Ethereum Virtual Machine
DAPP	Decentralized Application
RPC	Remote Procedure Call
LP	Liquidity Provider
ERC	Ethereum Request-for-Comments
MLM	Multilevel Marketing
ETH	Ether
TVS	Total Value Secured
DEX	Decentralized Exchange
CEX	Centralized Exchange



List of Notation

LP_{price}	Price of Liquidity Pool Token
A_{out}	Output Amount
A_{in}	Input Amount
C	Cumulative Amount
S	Stored Cumulative Amount
t	Time Interval
P	Price
Z	Data Type
TP	True Positive
FP	False Positive
V	Token Value
a	Amount of Token
r_0	Token0 Reserves
r_1	Token1 Reserves



Chapter 1 Introduction

With the rapid development and growth of distributed ledger technology (DLT) and the cryptocurrency market, various applications products built on the blockchain have flourished. Finance products, prediction markets, and Ponzi schemes show a new appearance to the public. Users and Funds increased dramatically. It has grown several hundred times from 600 million at the beginning of 2020 to a historical high of 180 billion in November 2021 [1]. This juicy target also attracts hackers, making 2022 as the year with the highest number, as well as the one with the most loss of attacks in Decentralized Finance (DeFi).

In order to enter the market quickly, many developers neglect basic software testing and quality assurance. Leaving the smart contract untested may lead to unexpected results or be vulnerable to attackers. Akutars [2], a Non-Fungible Token (NFT) project on Ethereum, launched with a flawed Dutch auction smart contract and permanently locked 11,539 Ethereum (ETH), worth 33.8 million USD at that time. The tragedy was made by simple condition check errors [3] in the smart contract source code that made the Akutars' project creator not able to withdraw funds from the smart contract.

Unlike traditional software, smart contract cannot be patched after deploying on blockchain. It makes smart contract testing imperative during development. Methods

of testing smart contracts can be classified into two categories: automated testing and manual testing. Automated testing employs tools to examine smart contract source code automatically to find defects or executed repeated tests for execution errors. It is especially valuable when tests are repetitive, challenged to perform manually. Property-based testing is a testing technique used in software development to automatically generate test cases, based on properties or invariants the software should uphold. Static analysis and dynamic analysis are two methods for executing property-based testing. Slither [4], SmartCheck [5], Zeus, [6], and Securify [7], are open-source static analysis tools. They examine the smart contract source code or bytecode to detect a range of security vulnerabilities in smart contracts, including integer overflow or underflows, access control, reentrancy [8], etc. Mythril [9], Echidna [10], Manticore [11] are dynamic analysis tools, which generate symbolic inputs or create inputs to a smart contract function to see if any execution trace violates specific properties with malformed and invalid data. Smart contracts rely on inputs provided by users to execute functions. Incorrect or malicious input to smart contracts may lead to unintended results or attacks. Smart contract attacks are often composed of malicious inputs and a series of function calls to the victim smart contract.

However, automated testing tools may have restrictions, such as missing bugs due to predetermined vulnerability patterns or generating numerous false positives. They

are not able to discover new vulnerabilities or business logic defects of an applications.

With the development of blockchain applications, smart contract project architecture and its business logic has become more and more complicated. A report [12] conducted by Chainalysis, a blockchain analysis firm, mentioned that not all of the attacks are what one may think of as hacks in the traditional sense. In some cases, attackers manipulate price oracles to drain the DeFi protocol of funds without taking advantage of an error in the protocol's code. Zou et al.'s [13] interview with smart contract experts exposed a lack of best practice writing reliable code. Finding code examples is especially difficult while developing new applications in smart contracts, although smart contract developers and auditors are both familiar with code-level vulnerabilities because of easy access to these analytical tools. Logic level vulnerabilities are still hard to find or even be found when attacks have happened. Euler Finance, a noncustodial lending protocol, was hacked for over \$195 million. The root cause of the attack was inconsistencies of users borrowed assets and collateralized assets which may be triggered by `donateToReserves()` function in *EToken.sol* [14]. The fatal vulnerability remained undetected since it was deployed to the Ethereum network, even though the code was audited by Sherlock [15], a security firm, until the attack had happened.

In recent years, smart contract researchers have devoted attention to DeFi security. Li et al. [16] systematically analyze and classify DeFi security issues according to its

vulnerability principle on different aspects, source data, private key management, and smart contract vulnerabilities, which may all affect the user's fund safety. Wang et al. [17] proposed an analysis framework that detects price oracle vulnerabilities via mutating transaction parameters. They discovered that five out of six known price oracle vulnerabilities, as well as 27 new price oracle vulnerabilities in DeFi Protocols. Wang et al. [18] presented BLOCKEYE, a real-time attack detection solution for DeFi projects on the Ethereum blockchain, which performs symbolic reasoning on the data flow and checks the possibility of state manipulation. Wu et al. [19] implemented a prototype named DEFIRANGER that detects direct and indirect manipulation on token price in DEX and DeFi apps by analyzing high-level DeFi semantics constructed from the token flow of real-world Ethereum transactions.

In this study, we are inspired by the past DeFi attacks on price oracle vulnerabilities and lack of tools or methods to test its ability to resist price manipulation. Although to the best of our knowledge there are no analysis tools that can detect price oracle manipulation vulnerability, we proposed a detection method on price oracle vulnerabilities by sending simulated message calls by skewing the liquidity pool reserves. We first filter out the transaction, which involves the Automated Market Maker (AMM) price-related function calls to liquidity pool smart contract internally to serve as our dataset. Then we run simulation tests with mutated state variables on the

price oracle to test its quote function. Compared with its original quote and mutated quote, we can distinguish which may be vulnerable to its project. We found 103 distinct price oracles that may be vulnerable by our method, and 38 of them were confirmed as true positives manually. We also tested our method on past price oracle attack incidents. Our method can detect parts of them and can prevent the attacks, while the other tools cannot.

The main contributions of this study are as follows:

- We proposed a detection method of price oracle manipulation vulnerabilities, which doesn't need information about price oracle contract address and referred liquidity pool address.
- Our method is also applicable to real time detection, which is also available to integrate with different malicious transaction alert systems.
- We reorganized and categorized the token flow and the definitions of DeFi actions.
- Researchers can design different simulated attack vectors according to our experimental environment and methodology.

The reminder of this study is organized as follows. Chapter 2 introduces fundamental background knowledge about Decentralized Finance (DeFi) and how to attack a project using price manipulation. Chapter 3 describes our method of how we simulate attacks and why we mutate blockchain state variables. Chapter 4 shows our

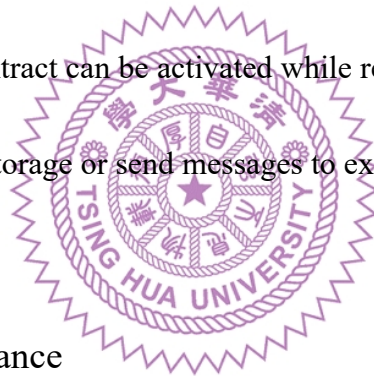
experimental setup and the results and restrictions of all experiments. In Chapter 5, we will additionally tackle the threats to validity and provide responses to various research questions. Finally, we conclude our study and discuss the future work in Chapter 6.



Chapter 2 Background and Related Works

2.1 Smart Contracts

Ethereum has two different types of accounts. One is the External Owned Account (EOA), controlled by private keys. Another is the Contract Account, controlled by its contract code. Anyone who knows the private key can create and sign a transaction. Smart contracts are self-executing, programmable digital agreements that run on the Ethereum blockchain. They are the fundamental building blocks of applications, such as decentralized finance (DeFi) and tokenization Decentralized Autonomous Organization (DAO). A contract can be activated while receiving inputs, allowing it to read and write its internal storage or send messages to execute another smart contract.



2.2 Decentralized Finance

Decentralized finance (DeFi) is a broad term referring to the ecosystem of financial applications and services built on blockchain technology. It aims to create an open, transparent financial system to everyone which is not controlled by any central authority, such as government, banks or institutions. Developers can create a variety of products, including lending and borrowing, insurance, spot trading, derivatives without carrying about the government regulations.

Decentralized Exchange:

Decentralized exchange is a trading platform built with smart contracts on blockchain that allows users to trade different tokens without a central authority. Users can trade digital assets directly with one another or with liquidity providers. Unlike centralized exchanges, it is permissionless to list trading pairs and will not place sanctions to users.

Automated Market Maker

Market Makers actively quote two-sided markets by providing bids and asks on centralized exchanges. They provide liquidity and depth to markets and profit from the difference of the bid-ask spread [20]. Centralized exchanges can ensure that counterparties are always available for all trades [21].

Automated Market Makers (AMM) was first introduced by Vitalik Buterin in 2017 [22]. It is an autonomous protocol that can price assets and provide liquidity based on a function or algorithm instead of by using a traditional order-book-style to match submitted bids and asks to execute trades. One of the most popular types of AMMs is the Constant Product Market Maker (CPMM) implemented in UniswapV1 [23] and UniswapV2 [24]. It uses a simple $x*y = k$ equation to set the relationship between the two assets' balances, x and y , held in the liquidity pools. Figure 1 shows that the condition of the liquidity pool reserves before the trade at T and becomes to T' after a

trade. Both T and T' satisfy the equation under ideal conditions without considering commission.

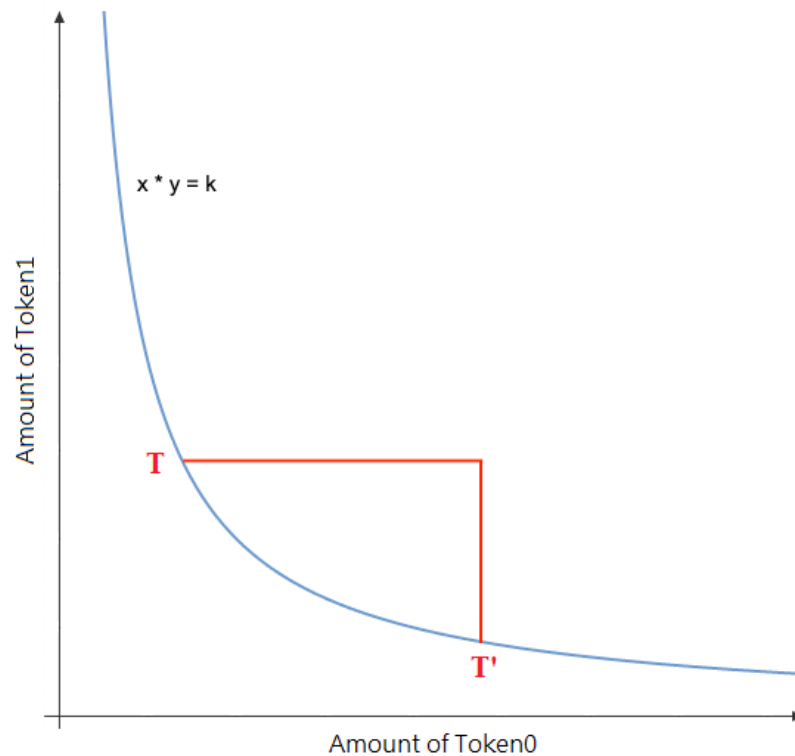


Figure 1: Graph of CPMM Model

There are also different AMM protocol designs to meet different use scenarios. Curve Finance modifies the AMM formula to make it better for creating stable pools that concentrate liquidity to provide a lower slippage and lower fee. Balancer took its AMM formula to the next level by making them more customizable. Balancer's weighted pools enable users to build pools with more than two tokens and custom weightings. Another major category is the Uniswap fork protocol; for example, Sushiswap and PancakeSwap which are all derived from Uniswap and rewritten with

different parameters.

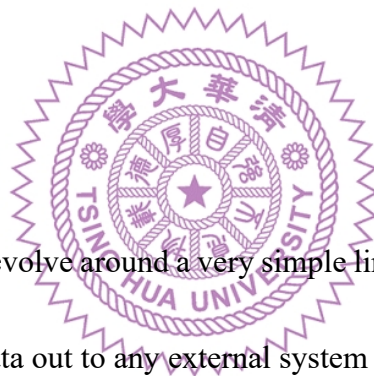
Oracle

Oracles are data feeds that source, verify and transmit external information to smart contracts running on the blockchain. It gives the smart contract the ability to execute off-chain data such as temperature, commodity price, etc. They are essential for various DeFi applications, like decentralized exchanges, lending platforms, prediction markets, and stable coins. Oracles differ based on the number of data sources, whether the trust model is centralized or decentralized, and how its architecture is designed [25].

Oracle Problems

The oracle problems revolve around a very simple limitation – blockchains cannot pull in data from or push data out to any external system as built-in functionality [26].

Regarding blockchain and smart contracts, the oracle problem involves the trustworthiness and reliability of oracles [27]. Curran [28] defined the blockchain oracle problem as the security, authenticity, and trust conflict between third-party oracles and the trustless execution of smart contracts. To the best of the author's knowledge, the problem with blockchain oracles is that they introduce a potential point of failure and vulnerability in the decentralized and trustless nature of blockchain technology. We list different aspects of this issue.



- Centralization Risk

While blockchains are decentralized by nature, relying on a single price oracle can lead to centralization. If an oracle becomes compromised, the smart contracts depending on it could receive inaccurate data, leading to adverse consequences for the entire ecosystem. Using multiple oracles to establish an oracle network can reduce the centralization risk and single point of failure.

- Data Manipulation and Attacks

Price oracles can be susceptible to various forms of manipulation. An attacker can utilize large capital to manipulate the asset price, then the oracle receives the incorrect data and publishes it to the oracle network. Oracle algorithms may possibly eliminate the data that was deviated from others and make the aggregated results stable. Pyth Network has experienced abnormal price quotations due to its algorithm weights all publishers equally, including abnormal precision outlier data [29], which has caused users to be wrongly liquidated.

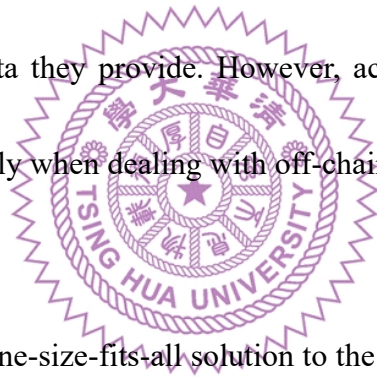
- Latency

Real-time price data is crucial for many DeFi applications, but delays in data transmission or processing can result in inaccurate or outdated information. Latency can be particularly problematic in fast-moving markets, where even a small delay can lead to incorrect pricing and forming an arbitrage opportunity.

Chainlink, for example, relies on the administrator to publish its aggregated results to the blockchain. If the blockchain is congested or halted, the new price cannot be published; the origin price becomes out-of-date.

- Reliability and Transparency

If an oracle goes offline or experiences technical issues, the smart contract relying on its data might not function correctly. This can lead to losses for users and affect the overall trust in the DeFi ecosystem. Trust in the data provided by an oracle is crucial. Price oracles need to ensure transparency in their operations, so users can trust the data they provide. However, achieving transparency can be challenging, particularly when dealing with off-chain data.



Although there is no one-size-fits-all solution to the oracle problem, one possible solution [30] includes the following: a decentralized network of oracles to increase data reliability and security, employing cryptographic techniques such as zero-knowledge proofs or secure multi-party computation to enhance data privacy and security, developing oracle-as-a-service platforms to simplify the integration of off-chain data into smart contracts, and leveraging game and token-based incentives to encourage honest behavior among oracle providers.

Flash Loan

Flash loan is a type of uncollateralized lending, in which borrows and repays need to be finished in the same transaction. Users can gain access to a very huge amount of capital without upfront deposit collaterals with fees as low as 0.3% or even zero fee. Flash Loan was first introduced in 2018 by Marble, an open-source bank on the Ethereum blockchain [31]. It was often used in arbitrage trades on different decentralized exchanges daily. However, flash loans are known by the public as a tool used by hackers to attack certain DeFi projects.

2.3 Price Oracle Manipulation

Price oracles are expected to provide reliable and accurate price data to the smart contract and to the decentralized applications (DAPPs) in the blockchain. If there are vulnerabilities in the price oracle systems, a malicious user may utilize it and manipulate the price of cryptocurrencies or other digital assets to make profits from the DeFi project. In Eskandari et al. 's research [32], most oracles share similar high-level architecture, including initial data, data sources, data feeders, aggregation, and dispute phase. We will show some examples of how to fluctuate the price by manipulating its price oracle on different data sources, DEX liquidity pools, and CEX spot prices.

Example 1. Suppose a decentralized finance (DeFi) over-collateral lending project relies on an on-chain price oracle to obtain the price of cryptocurrency Token A. Its price oracle uses a single liquidity pool to calculate Token A's price. Users deposit Token A into the project's smart contract to borrow another cryptocurrency, Token B. User's borrowing limit depends on the value and borrow factor of its collateral, Token A. Alice, a malicious actor, notices that the project allows users to deposit \$ETH and borrow \$DAI, which uses a faulty price oracle to calculate \$ETH price on a single liquidity pool ETH-USDT. She decides to manipulate the price oracle using flash loan to make profit from the project. To do this, she follows these steps:

1. She first uses flash loan to borrow \$ETH and \$USDT from Aave. Then she deposits \$ETH to the victim project as collateral. In this case, Aave merely serves as a provider of flash loan services.
2. Then she uses a large amount of \$USDT to buy \$ETH at the liquidity pool that the faulty price oracle used as its data source. After Alice's trade, the amount of \$USDT in the liquidity pool increased, and the amount of \$ETH in the liquidity pool reduced accordingly. Therefore, the price of \$ETH quoted in \$USDT soared.
3. The victim project uses the spike price of \$ETH to calculate Alice's deposited collateral. It allows Alice to borrow a much larger amount of

\$DAI than she would have been able to under normal circumstances. Alice then can take all available \$DAI and leave her collateral in the victim project.

4. Alice then sells all the \$ETH she bought at step 2 to get \$USDT back and to repay the flash loan from Aave. The liquidity pool reserves of both token \$ETH and \$USDT return to its original amount without calculating trading commissions. The price query from the vulnerable price oracle return to its normal price also.
5. However, Alice's account on the victim project became underwatered. She uses the lesser value of \$ETH to deceive the victim project to borrow a higher value of \$DAI. The bad debts will be borne by all depositors who provide liquidity to the victim project.

This is a simple example of deceiving the platform to miscalculate the value of the users' collateral. There are also more complex examples, such as in the case where other flawed functions are combined to mislead the user's account health factor [33]. This health factor is the ratio between borrowing capacity over outstanding debts, all of which can be considered as broad definition of price oracle vulnerabilities.

Example 2. GMX, a popular decentralized perpetual futures trading platform on Arbitrum and Avalanche chains, suffered a price manipulation exploit on the

AXAX/USD trading pair on Sep. 19, 2022. The attacker managed to make off with around \$565,000 profit from the platform liquidity providers. GMX has a fixed futures price for assets fed by an oracle that allows users to trade with zero slippage of perpetual trades. The price oracle of GMX is closed source, with its only disclosures being that it will aggregate data from Chainlink oracles and multiple centralized exchanges spot price to determine its futures price on the platform. The attackers manipulate the \$AVAX spot price on centralized exchanges started at 09:15 (UTC +8) as shown in Figure 2. Then opens large size of long or short position on GMX AVAX/USD market without affecting the market price, after the price oracles update the platform with the manipulated price. The attacker manipulates the centralized exchange spot price again which triggered the GMX price oracle to update its futures price again. The attacker now closes its positions on GMX to realize profit. Utilizing the high leverage and zero slippage on GMX, the attacker opens over 220,000 AVAX unit futures, in which the nominal value is more than 4 million, and earn approximately \$100,000 in each circle. The lowest spot price of each cycle is not less than 18.3 USD, and the highest is not more than 18.78 USD, which is only a 2% fluctuation.



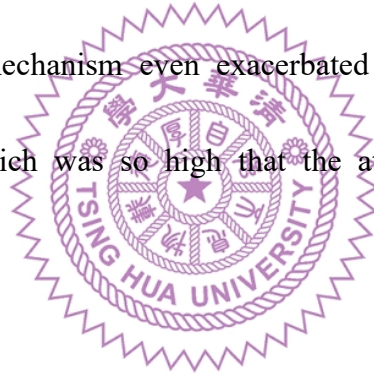
Figure 2: Market Manipulation on CEX

2.3.1 Preventions From Price Oracle Attack

Project Developers are familiar with vulnerabilities like reentrancy, arithmetic overflow, and underflow in Solidity. Vulnerabilities in the price oracle used in smart contracts are not something that is often considered. Centralized oracles such as Chainlink [35] do not update the price very often. There are two parameters that will trigger Chainlink nodes to update price: One is deviation, and the other is time interval. Take Chainlink's ETH/USD for example; the deviation parameter is 0.5%, and the time interval is 1 hour.

Decentralized oracles which rely on the on-chain data may reflect on the on-chain data change too quickly [30]. Developers can add some artificial delay to increase the difficulty to manipulate the price oracle. Uniswap V2 and V3 provide built-in time-

weighted average price (TWAP) and time-weighted average tick (TWAT). In Uniswap V3, tick is the smallest unit of price, so the price is not linear; similar to a ladder in a traditional order book. Developers can use any Uniswap liquidity pool as the price oracle data source. The disadvantage, however, is that it may not respond quickly enough to a high volatility market, and it only works for tokens which already have liquidity and is large enough to resist long-term price manipulation. Rari Capital's Fuse permissionless pool attack is an apt example illustrating how liquidity-insufficient pools used as the price oracle data feed can be easily manipulated. Uniswap V3's geometric mean TWAP mechanism even exacerbated the price manipulation and skyrocketed the price, which was so high that the attack's collateral value even overflowed the UI [36].



2.4 ProMutator

ProMutator [17] was represented by Wang et al., which tests the AMM-based price oracles by mutating transaction input parameters and return values of price-related functions. An example function call flow of a project utilizing a liquidity pool as its price data feed is shown in Figure 3. Steps 1 and 3 are where ProMutator mutated the value of a real transactions. They will replay the transaction with mutated value and then judge its new state with its detection rules. Our proposed method overwrites price

data at step 2 to test whether the result of the project's smart contract is manipulated or not. Mutating values at different step may need different detection rules and may have different scalability. Designing a new detection rule is more difficult in ProMutator. It will be easier to design different simulation scenarios as in our proposed method at step 2.

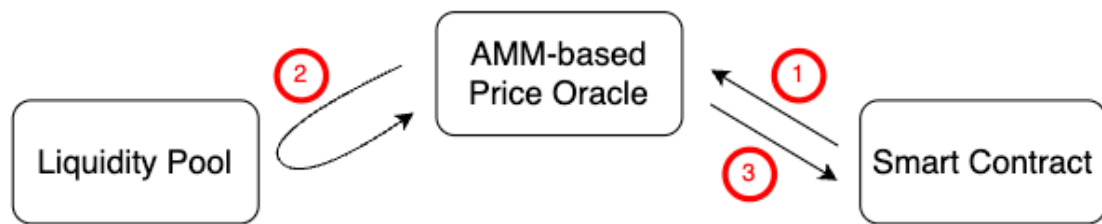


Figure 3: Price Oracle Function Call Flow in a Transaction

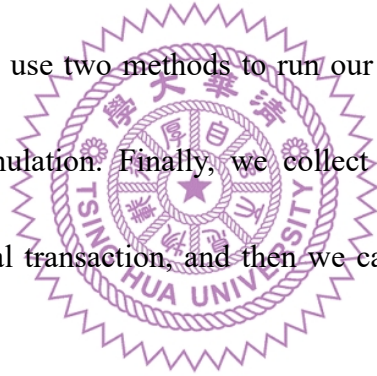


Chapter 3 Methodology

Ethereum Virtual Machine (EVM) is a state machine, which is a decentralized computer designed to execute smart contracts on the Ethereum blockchain. In the context of the EVM, a state machine refers to a system that changes its states based on a set of predefined rules and inputs. The EVM processes transactions and is responsible for maintaining the state of the Ethereum network. The state of the Ethereum blockchain consists of a collection of accounts and their associated data, such as balances, storage, and contract code. When a transaction sent by external actors is executed, the EVM processes it and updates the state of the network accordingly.

We can simulate a transaction locally, within an independent node's EVM instance, without sending it to the Ethereum network. It allows us to preview whether there has been success or failure of the transaction and whether data was returned by smart contract functions, and to understand how the transaction might affect the state of the Ethereum blockchain. We can also simulate malicious transaction or mutate normal transaction parameters to test smart contracts. There are some similar project and research that analyze and broadcast transactions locally to simulate the next state of the blockchain. Wang et al. present ProMutator [17], which is a security analysis framework that detects price oracle vulnerabilities by simulating mutated transactions. KillSwitch [37], a self-service SaaS platform developed by PeckShield, monitors and

simulates pending transactions in the mempool to prevent DeFi projects from an exploitation with a just-in-time protocol pause or an emergency fund withdrawal. In this chapter, we demonstrate our proposed method: Foundry Forge Test and message call simulation. The flowchart of our experiment is shown in Figure 4. In the first step, we retrieve transactions which contains target AMM price-related functions. Second, we parse the invocation flow of the transaction to get the caller and callee of our target functions. Meanwhile, we prune transactions with predefined rules in Table 5. Some transactions may contain price-related functions in internal function calls but do not act as a price oracle. Then, we use two methods to run our experiments: Foundry Forge Test and message call simulation. Finally, we collect the experimental result and compare it with the original transaction, and then we can distinguish which one has potential vulnerability.



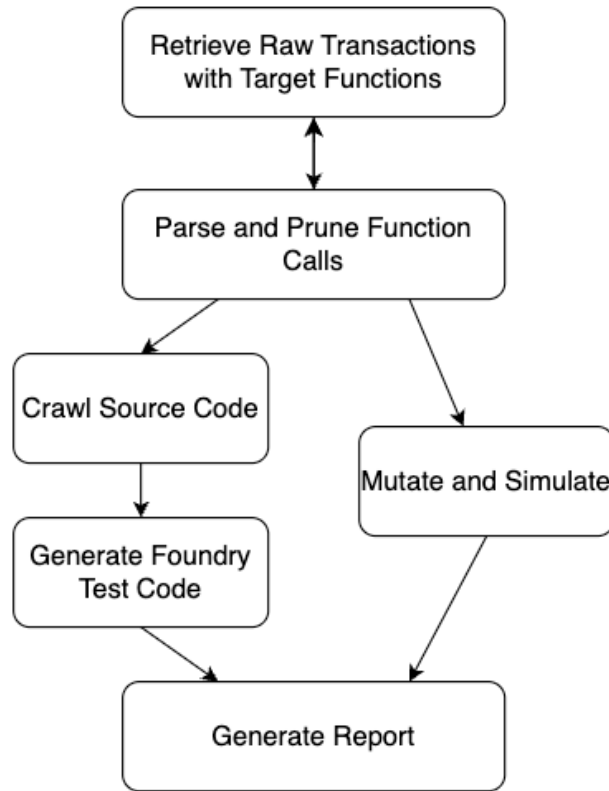


Figure 4: Flowchart of Proposed Method

3.1 Foundry

Foundry is a powerful smart contract development framework designed to make the process of creating and developing smart contracts and managing decentralized applications (DAPP) more efficient and user-friendly. It is designed to support multiple programming languages, such as Solidity, Vyper, and Rust for smart contract development. Developers can interact with the blockchain from command-line or via solidity-scripts, which makes unit testing more easier than other development frameworks like Hardhat and Truffle. Foundry is consisted of three main components: Forge, Anvil and Cast.

Forge

Forge is the main part of Foundry. It is a command-line tools that ships with Foundry. Developers can use forge to build and compile the smart contract projects. It can automatically switch the solc compiler version [38] specified in the smart contract source; also it can rearrange third-party libraries in the project directory which makes importing GitHub repository or inheriting Truffle or Hardhat projects more efficiently. Forge Standard Library is a built-in collection of helpful smart contracts for writing unit test case in Solidity. Developers can write unit test case in Solidity, while Truffle and Hardhat needs developers to write test script in JavaScript.

Anvil

Anvil is a local testnet node shipped with Foundry. It can reconstruct blockchain from a snapshot it gets from an archive node or construct a local blockchain. Developers can deploy their smart contract and interact with it on Anvil.

Cast

Cast is Foundry's command-line tool for performing Ethereum Remote Procedure Calls (RPCs). Developers can make smart contract calls, send transactions, or retrieve any type of chain data from command-line.

Figure 5: Foundry Test Example

nal source code and write a
 npile it with the Forge comm
 d compile the unit test source
 each test and emit an error me

Figure 6: Execution Traces from Foundry

Restrictions on using Foundry:

However, we have met some restrictions using Foundry to run our experiments.

We have to change our setups to use the message call simulation.

- Lack of Source Code:

Some smart contracts do not verify on Etherscan, which means that the smart contract deployer does not upload its source code and remains secret. We can only get the smart contract's bytecode which is compiled from its source code. If we can't get its source code, we cannot test it using Foundry.

- Problems of the Solidity Version:

The Foundry Standard Library is a crucial collection of smart contracts sets designed for testing and debugging purposes. However, it requires a minimum of the Solidity version 0.6.2. Solidity does not support backward compatible completely.

Some smart contracts or imported libraries, which are written in the older Solidity version, is not compatible with the Foundry Standard Library. We are not able to compile it nor write a unit test case to test it.

- Missing Library Path:

Some smart contract source codes verified on Etherscan do not include the directory of the import library. Etherscan only accepts source code file; developers cannot upload the entire project. With a lack of directory information, Foundry are not

able to compile the entire source code. Although we can add the import library path manually into the source code, it will take us lots of time inspecting each smart contract source code.

3.2 Simulation Function Calls

Ethereum JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol which defines data structures and rules around their processing. Every execution layer client implements a JSON-RPC specification which supports standard collection of methods. It is the canonical interface between users and the blockchain network [34]. Users interact with the execution layer by sending requests through specific JSON-RPC methods [39]. All methods are grouped into several categories depending on their purpose. Because of its efficient state storage, faster synchronization, and the additional useful methods it has implemented [40], we choose to run Erigon as our execution layer client.

Transactions are cryptographically signed instructions initiated from an externally owned account (EOA). Each time a transaction occurs, it alters the state of the EVM. There are a few different types of transactions on Ethereum. Firstly, a regular transaction is a transaction sending Ether, a native cryptocurrency on the Ethereum network, from one EOA to another EOA. Secondly, the contract deployment transaction

is a transaction without a "to" address, where the data field is used for the compiled contract code. The third type of transaction is an EOA that interacts with a deployed smart contract. The "to" address in these types of transaction is the contract address. A transaction which interacted with a smart contract often contains internal function calls to different smart contracts. Internal calls refer to the interactions between different smart contracts within the same transaction. These calls occur when one smart contracts caller invokes a function in another contract callee. Figure 7 shows the execution traces of a transaction [45] visualized by Tenderly [46] in which a user borrows 3,000 USDC from Aave V3, and calls other smart contract functions internally, including some AMM price-related functions to calculate its collateral value and debt value.

To get the detailed information of a transaction, we need to replay the transaction in the Ethereum Virtual Machine and collect the execution data during the process. The lowest level of transaction traces that that execution layer node can generate are raw EVM opcode traces. The tracing methods accept an optional tracer parameter that defines how the data that is returned to the request should be processed. The default raw opcode traces are useful, but the returned data is low level and can be too extensive to read. There are a set of built-in tracers that can be used in tracing methods, which do some specific preprocessing on the trace data to return the readable data format from the node. It is also possible to provide a custom code that hooks to events in the EVM

to process the data in a user-defined format.

Both the `debug_traceTransaction` [41] and the `trace_transaction` [42] methods can generate the EVM trace, but the `debug_traceTransaction` method returns lots of low-level opcode that can be used to show what happens step by step during the process and the reason for the failed transaction [43]. The `trace_transaction` method only accepts the transaction hash as its only argument, and will aggregate all generated data and return it as a large JSON object. We choose `debug_traceTransaction` and write our custom EVM tracer [44] to parse the transaction and enable it to check transaction – which contains our target price-related function signature – and which is inspired by ProMutator [17].

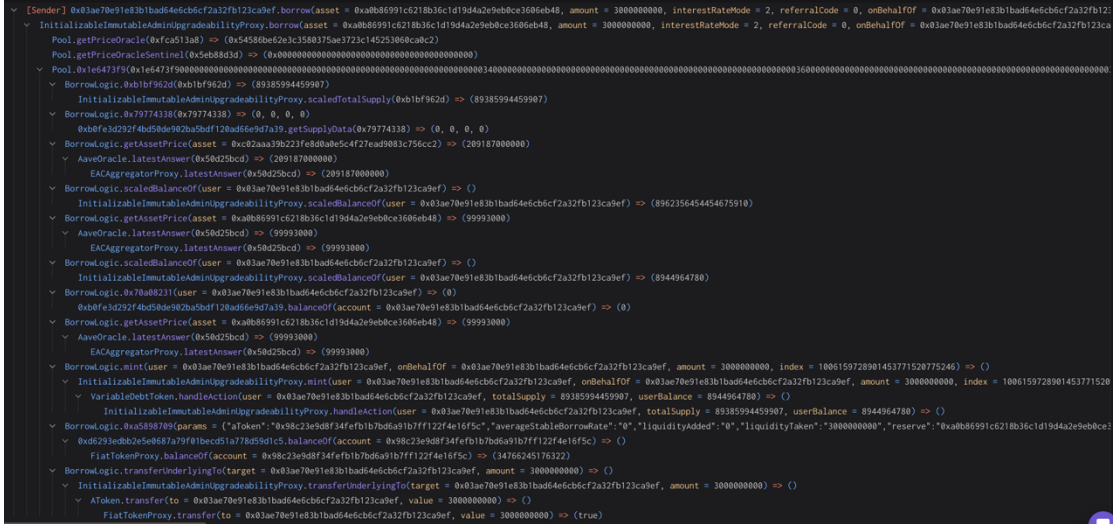


Figure 7: Execution Tree of Transaction

After getting the execution trace of a transaction, we retrieve the internal function calls which are price-related functions and its caller, function arguments, and returned

results. EVM is a state machine, and if we have the same state and transactions, we can get the same next state. So, we can simulate a transaction with a mutated parameter or override the EVM state storage slot to get a different EVM state. By analyzing the receipt of the mutated transaction and the identical transaction, albeit processed with different storage slot values, we can filter out which AMM-based price oracle may be affected by malicious attacks.

We use `eth_call` to call the AMM-based price oracle function with a mutated state variable to simulate a manipulation attack. It is a commonly used method call that can read from the blockchain and which includes executing the smart contract, but does not broadcast the transaction to the blockchain. For read-only functions, it returns what the read-only function returns. For functions that change the state of the contract, it executes the transaction locally and returns any data returned by the function. When we call the target price oracle with `eth_call`, we can add the state override set option to modify the state variables stored in the blockchain before executing the calls. If the result returned by target price oracle has an abnormal difference from the original transaction result, we will mark this price oracle as potentially flawed.

Some price oracle implementations store the price calculation results in its storage slot, unlike others that return calculation results through functions. Therefore, we need to know in which storage slot the result is stored. We use `eth_createAccessList` in Figure

8, an RPC method which creates an EIP2930 type access list that contains all storage slots and addresses read and written by the transactions. Next, we compare the access list with another RPC method, `trace_call` with `stateDiff` option of trace, which executes the given call and returns a number of possible traces. It is helpful for debugging transactions and analyzing state changes due to a transaction. Comparing with the access list we get from `eth_createAccessList` and `trace_call`'s `stateDiff`, we can distinguish which storage slot is to be read or to be written by the transaction execution.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "accessList": [
      {
        "address": "0x0d4a11d5eeaac28ec3f61d100daf4d40471f1852",
        "storageKeys": [
          "0x0000000000000000000000000000000000000000000000000000000000000000",
          "0x0000000000000000000000000000000000000000000000000000000000000009",
          "0x000000000000000000000000000000000000000000000000000000000000000a"
        ]
      },
      {
        "address": "0x833e440332caa07597a5116fbb6163f0e15f743d",
        "storageKeys": [
          "0x6200d9c0606964564958e1edf0b5d67e485691a223ce33e3267306cb461a390f",
          "0x1c390d3e57c185a5783a278f6af77cc49c9868812458fed2a4652ed2147410cb",
          "0x1c390d3e57c185a5783a278f6af77cc49c9868812458fed2a4652ed2147410cc",
          "0x0bcac52f91268e9e3558903845f95414f43bac88dcf2c02b8a5a2f6a173b45e7",
          "0x6200d9c0606964564958e1edf0b5d67e485691a223ce33e3267306cb461a390e"
        ]
      },
      {
        "address": "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2",
        "storageKeys": [
          "0x248dd7ef3a8494532f906b26cd4f09639b1e8249264b85f51a3c22819f7d433b"
        ]
      },
      {
        "address": "0x534f2675ff7b4161e46277b5914d33a5cb8dcf32",
        "storageKeys": [
          "0x0000000000000000000000000000000000000000000000000000000000000000"
        ]
      }
    ],
    "gasUsed": "0x15261"
  }
}
```

Figure 8: Example of `eth_createAccessList()`

We record the read storage slot list and check whether the location is the same as the state variable in the associated liquidity pool smart contract. If the storage slot is the same, we can infer the data source of the price calculation and the result's storage slot

without knowing the function name in the price oracle source code. Then we can manipulate the reserves in the liquidity pool by mutating the value of the state variable to simulate a possible attack. If the results of the price calculation changed then it means that the price oracle can easily be deceived and found by our method. The flowchart of two different design methods of the price oracle is shown in Figure 9.

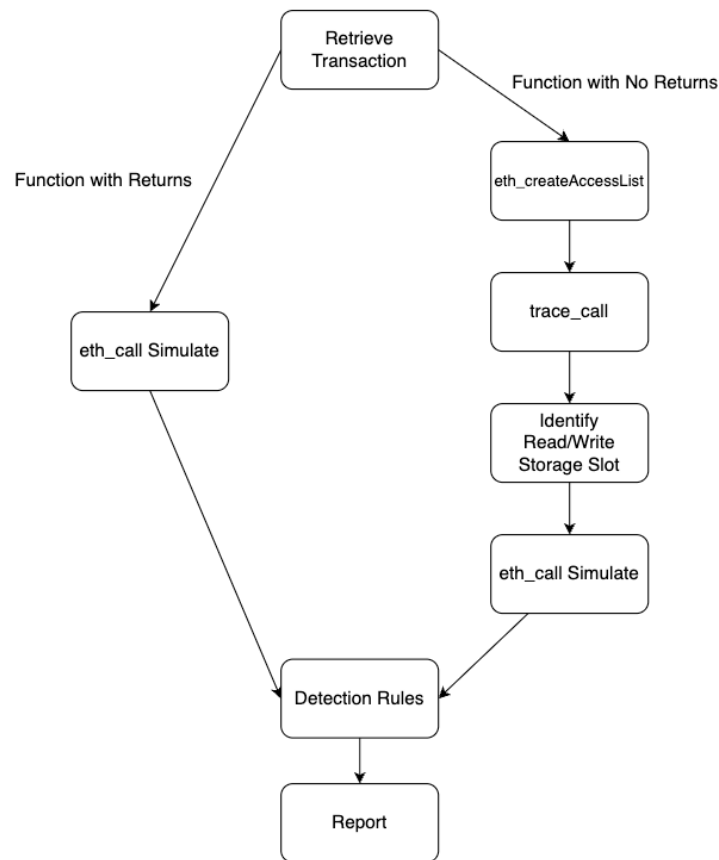


Figure 9: Flowchart of Simulation Function Calls

3.2.1 Warp Finance Incident

Warp Finance is a decentralized finance platform that aims to create a use case for unused LP (Liquidity Provider) tokens by allowing them to be used as collateral for

borrowing and to maximize the users' capital utilization. Lenders provide \$DAI, \$USDC or \$USDT to earn interest from borrowers. However, the implementation of the price oracle function used to calculate the value of collateral LP tokens is flawed, which causes a nearly 8M of financial loss. The root cause is that it uses a time-weighted average price (TWAP) to calculate the asset price, it still uses real-time assets balances in the liquidity pool to calculate the LP token value, as shown in Equation 1.

$$LP_{price} = \frac{r_0 \times TWAP_0 + r_1 \times TWAP_1}{LP_{amount}} \quad (1)$$

Liquidity pool reserves can be easily manipulated using flash loan by a malicious user. Although the flaw is simple and easily distinguishable at a glance, it still cannot be detected by auditing tools, such as slither [4], and echidna [10], which is commonly used by auditing firms. We use slither to perform static analysis on the smart contract, shown in Figure 10. It identified some flaws in the smart contract, but it could not detect the logic defects in this incident. However, this logic flaw can be detected by using our approach, i.e. by parsing the function calls that are price-related function first, then overwriting its state variables associated with them in order to simulate a manipulation of the reserves. If the result of the same function call has been changed, it indicates that there is a potential flaw. In Figure 11, the invocation flow of white background is the transaction's execution trace, and it visualizes the call graph and the result of each function. We screenshot the flawed price oracle partially from the attack transaction to

demonstrate and compared it to our experimental result which is in the black background. Our results demonstrate that by intentionally skewing the reserves of the liquidity pools, the flawed price oracle, which utilized these pools as its data sources, can be easily manipulated to return inaccurate prices to its protocol. Our method effectively detects vulnerabilities of this nature.

```
vincent@huangbaiweideMacBook-Pro WarpFinance % slither 0x4A224CD0517f08826608a2f73bf390b01a6618c8 --exclude-low
INFO:Detectors:
UniswapV2OracleLibrary.currentBlockTimestamp() (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#1000-1002) uses a weak PRNG: "uint32(block.timestamp % 2 ** 32)" (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#1001)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-prng
INFO:Detectors:
UniswapV2Router02.removeLiquidity(address,address,uint256,uint256,uint256,address,uint256) (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#499-515) ignores return value by IUniswapV2Pair(pair).transferFrom(msg.sender,pair,liquidity) (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#509)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
UniswapV2Library.getAmountsOut(address,uint256,address[]).i (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#909) is a local variable never initialized
UniswapV2Router02._swap(uint256[],address[],address).i (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#609) is a local variable never initialized
FixedPoint.mul(FixedPoint.uq112x112,uint256).z (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#966) is a local variable never initialized
UniswapV2Router02._swapSupportingFeeOnTransferTokens(address[],address).i (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#718) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
UniswapV2Router02._addLiquidity(address,address,uint256,uint256,uint256,uint256) (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#429-456) ignores return value by IUniswapV2Factory(factory).createPair(tokenA,tokenB) (crytic-export/etherscan-contracts/0x4A224CD0517f08826608a2f73bf390b01a6618c8-UniswapLPOracleFactory.sol#439)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Figure 10: Slither Result on Warp Finance Oracle [55]

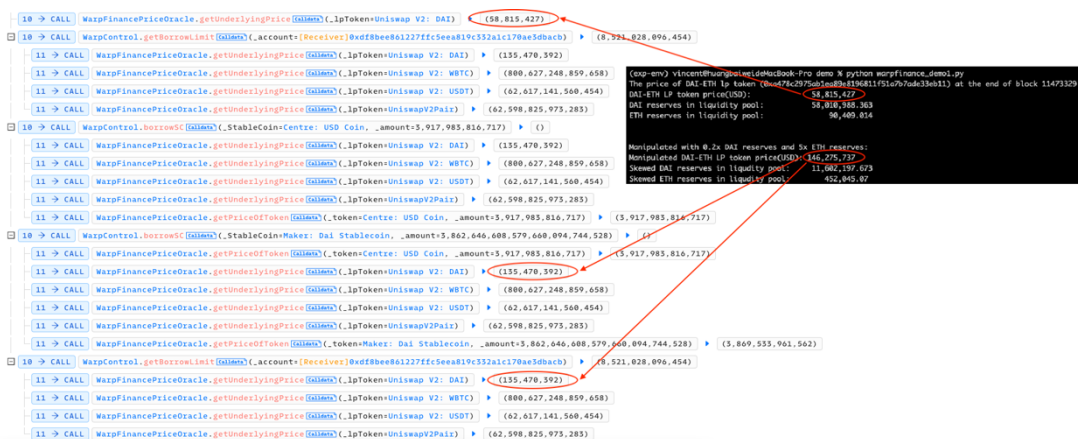


Figure 11: Comparison of Attack Transaction Invocation Flow and Test Result

3.2.2 Inverse Finance Incidents

Inverse Finance was exploited twice in 2022. The first attack happened on Apr 2nd, which caused a 14.8 million loss. The attacker spent 500 \$ETH, which was worth 1.7 million, in exchange for \$INV In order to manipulate its price of it on Inverse Finance. The root cause is the `current()` function, the price calculation logic of Inverse Finance in the Keep3rV2Oracle contract, which uses the SushiSwap liquidity pool as its data source. It first calls `price0CumulativeLast()` and `price1CumulativeLast()` to get cumulative amount of liquidity pool reserves of previous block, and then it computes with latest cumulative amount that was stored on price oracle to get the \$INV time-weighted average price. It is a common and convenient way to calculate the time-weighted average price like this, however, the `current()` function has a flaw in that it allows the calculation with a small amount of elapsed time value. The equation of calculating the ratio between input amount and output amount is shown below:

$$A_{out} = \frac{A_{in} \times (C - S)}{t \times 1e10} \quad (2)$$

In Equation 2, C indicates the amount accumulated to the previous block, S indicates the latest stored accumulative amount at the price oracle, and t indicates the time difference between C and S . If the time difference is one block in t , which means it manually stored the accumulative amount to S , then calculate the price at the next block. The equation can be simplified as below:

$$A_{out} = \frac{A_{in} \times [(S + P_0 \times t_0) - S]}{t \times 1e10} \quad (3)$$

$$A_{out} = \frac{A_{in} \times P_0}{1e10} \quad (4)$$

In Equation 3 and Equation 4, P_0 indicates the price that calculates at the current block and t_0 indicates the time difference between C and S , which is equal to one block. The calculation of P_0 is based on the referred liquidity pool 's previous reserves. Therefore, the price from the oracle in the current block will be the price that is manipulated from the previous block. In summary, if the time interval between manual updates of the cumulative value in the price oracle and price calculation can be short enough from preventing a third person to arbitrage the inflated price, then the time-weighted average price oracle is no longer effective in resisting manipulation of the reserve in the liquidity pool.

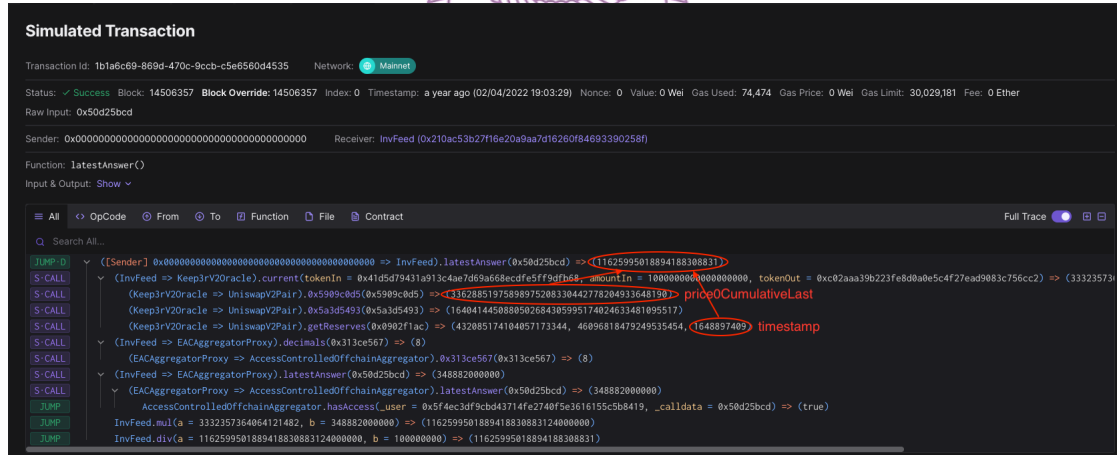


Figure 12: Simulation of Manipulating TWAP Price Oracle in Inverse Finance

In Figure 12, we demonstrate a simulation showing that it is possible to detect the

price oracle vulnerability by our method. We get all the involved storage slots first when calculating the asset price at Keep3rV2Oracle.sol [47], then we overwrite the cumulative value and timestamp in the referred liquidity pool and observation value in the price oracle to simulate both the malicious amount swap and the manual update that the attacker constructed before the exploitation. The manipulated price 11625.99 \$USD was 30 times higher than its original price of 394.23 \$USD at the block 14506356. We only overwrite the price0Cumulative and timestamp in Figure 12 because the rest does not affect the price calculation in this situation.

3.2.3 Cheese Bank Finance Incident

Cheese Bank Finance also encountered an attack similar to that of Warp Finance. Both manipulated the flaws in the pricing of LP collateral to over-borrow, causing bad debts for the platform. The difference is that the Cheese Bank Finance have different implementations of the price oracle. Cheese Bank implement implements separate functions to quote two types of asset prices, token prices and LP token prices. The price of each asset is stored in the storage of the smart contract, so it needs to be updated regularly to prevent it from being outdated. The `refresh()` is a write function which does not return the results; it updates its storage value with the new price. Therefore, we are not able to compare the function returned result like we used in detecting Warp Finance

vulnerability. We record all storage slots read and written by the price oracle function by `eth_createAccessList`, an RPC method. Then we compared it to `trace_call`, an RPC method which executed the given call or transactions and returns a number of possible traces, with the `stateDiff` config parameter. It provides information on the altered parts of the Ethereum state made due to the execution of the transactions. From these two pieces of information, we can identify whether the function involved is a read function or a write function. We mutate the state variable to read, compare it with the state variable to update the original transaction state, and then we confirm whether the price oracle in the Cheese Bank can be manipulated easily by manipulating the liquidity reserves.



3.3 Generating Mutate Value

Echidna [10], a Haskell program designed for fuzzing and property-based testing of the Ethereum smart contract, takes the smart contract and a list of invariants as input. It generates random sequences of calls and parameters to the contract and checks if the invariants hold. However, we don't need massive random parameters in our experiment. We mutate the smart contract storage slot value to simulate a different state of EVM. But not every state can be reached by a certain state. Figure 13 shows possible states of an UniswapV2 pool. There are only three ways to alter the pooled assets amount. Assets

reserved in the liquidity pool should follow the constant product market maker mechanism [23], which means that the product of the amounts of two pooled assets need to be the same, excluding the protocol fee if a swap happens. Adding liquidity and removing the liquidity amount should follow the ratio of the current amount of pooled assets.

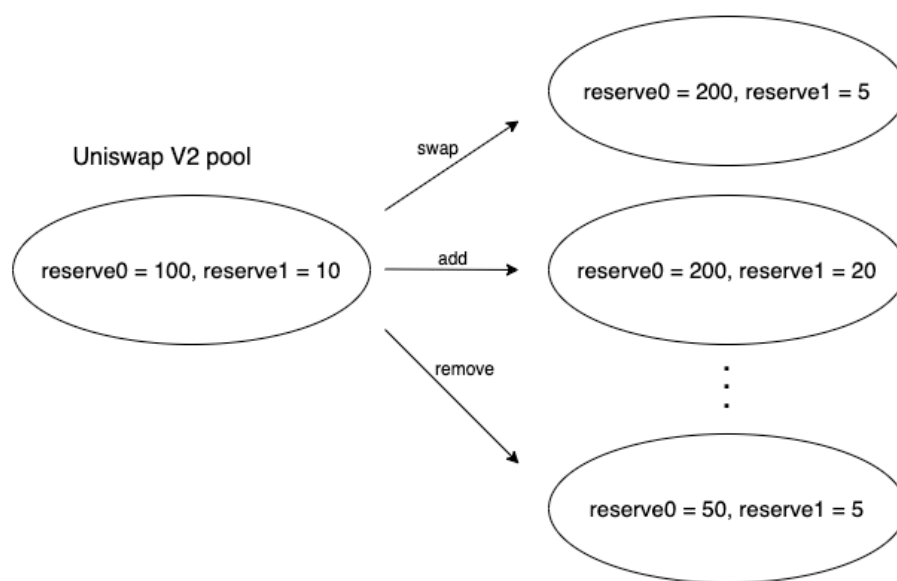


Figure 13: Possible State of an UniswapV2 pool reserves

We can overwrite the values in the contract address 's storage slot to modify the smart contract's state variables. This means that we can modify the assets amount in the pool without sending a transaction in our experiment. Before overwriting the value in storage slot, we need to calculate each variable's storage slot. According to the rules [48], different data types in Solidity have different ways to calculate the storage slot

index in Table 1. Data is stored contiguously item after item starting with the first state variable, except for the dynamic array and mappings. Therefore, different AMM protocols that may store pool asset reserves may store them in different storage slot. Solidity compiler supports generating the storage slot index in the JSON output after version 0.5.13, so we have to calculate AMM protocol 's state variable storage slot index ourselves beforehand if the source code uses older Solidity version.

Table 1: Summary of Calculating Storage Slot of Different Data Types

Data Type	Declaration	Value	Location
Variable	$Z \ x$	x	x 's slot
Fixed-size Array	$Z[n] \ x$	$x[n]$	$(x$'s slot) + $n * (\text{size of } Z)$
Dynamic Array	$Z [] \ x$	$x[n]$ $x.length$	$keccak256(x$'s slot) + $n * (\text{size of } Z)$ x 's slot
Mapping	mapping ($T1 \Rightarrow T2$) x	$x[key]$	$keccak256(\text{key}, (x$'s slot)

3.4 Prune Function Calls

In our datasets, we select multiple price-related functions in different AMM protocols as our target function. The wider the range we select, the higher the chances of including transactions with vulnerable price oracle within our dataset. However, this implies that we would encompass numerous activities associated with price-related interactions, but not specifically tied to price oracle transactions, within our dataset. So we prune transactions which are normal swap or other DeFi actions that are not associated with the price oracle before we run through our experiment. We took Siwen

Wu’s definition of DeFi actions [19] and then adjusted and expanded the scope of Wu’s original definition to cover a wider range of DeFi actions. Since we only need to identify different actions in the external or internal transactions, we parsed the token flow of the transaction to analyze the actions. The following are examples we found that do not conform to S. Wu’s definition, and we have modified the definition based on these examples in Table 2.

Example 1 Multi-hop Swap

It is very common to use `price0CumulativeLast()` and `price1CumulativeLast()` in the UniswapV2Pair contract to implement a TWAP price oracle [49]. Nevertheless, a simple swap transaction may also call price-related function to record price or access the liquidity pool balances. For example, Figure 14 illustrates the token flow of the transaction [50], which interacts with price-related functions from two liquidity pools, accessing liquidity pool cumulative prices and balances. In S. Wu’s definition, the sender sends a token to the receiver, then the receiver sends another token back to the sender to complete a swap. However, it may fail in a multi-hop swap. The first receiver, which is a liquidity pool smart contract, will send the second token to another liquidity pool smart contract to start a next swap, as shown in Figure 14.



Figure 14: Token Flow of a Multi-hop Swap Transaction

Example 2 Add Liquidity and Stake Token

Uniswap V3 allows users to add liquidity on the single side asset only, as compared to Uniswap V2, where users must add to both sides' assets in accordance with the current liquidity reserves ratio. A deposit or stake token to a smart contract does not always mint an underlying token as a receipt. Some smart contracts may only record the user's deposit amount on its storage slot.

Example 3 Customizable Token

The ERC-20 (Ethereum Request-for-Comments #20) token standard describes the methods and the event log that an Ethereum token contract must implement. Some of these rules include how the tokens are transferred, how the tokens are approved, and how to access the address's balance of the tokens. A smart contract that meets the requirements can be called as an ERC-20 fungible token, and once deployed, it will be

responsible to keep track of the deployed tokens on Ethereum. Deployers can implement additional functions in its ERC-20 token to achieve various functionalities which may make our definition becomes invalid. For example, it is common for tokens to have transfer taxes, which means that when there are transfers of tokens from a sender to a receiver, a certain proportion amount of token will be transferred to zero-address for burning or to a third-party address for further usage as a transfer tax. We exclude dividend tokens, i.e., tokens that can share a transfer tax or a buy-sell tax on another user's transaction, which may contain multiple transfers (T) or transfer burns (TB) within the internal transaction. We also exclude pyramid-scheme tokens, one of a multilevel marketing (MLM) token, where the distributor can earn extra profits from their downlines. The transfer of a pyramid-scheme token may contain multiple transfer (T) in its internal transaction to transfer commission to uplines. If we incorporate these special tokens into our definition of DeFi actions, our definition will become incredibly complex.

Furthermore, we did not take scam tokens and scam transactions into account. Zero-value token transfer attack [51], for instance, is the recent rise of a scam transaction, in which the attackers send a following zero-amount token transfer transaction to deceive victims that they are interacting with a familiar address, with the same first and last few characters on the address. Last but not least, tokens that

implement malicious functions or contain security vulnerabilities exploitable in the contract source code can bear resemblance to scam tokens. According to Xia. et al. ‘s research [52], roughly 50% of the tokens listed on Uniswap are scam tokens. We disregard it in our preprocessing of the dataset. There are also some tokens that intentionally conceal their tracks, or tokens that do not emit event logs on actions that does not fully comply with the ERC-20 protocol, which makes it difficult or impossible to track.



Table 2: Definitions of DeFi Actions

DeFi Actions	Symbol	Attributes	Conditions	Assignments	Description
Transfer	T	sender, receiver, amount, asset	T.sender!=T.receiver and T.spender!=zero-address and T.receiver!=zero-address and T.amount>0	sender=T.sender receiver=T.receiver asset=T.asset amount=T.amount	An address (sender) transfers certain amount (amount) of asset (asset) to another address (receiver)
Transfer Mint	TM	sender, receiver, amount, asset	TM.sender==zero-address and T.amount>0	Sender=T.sender Receiver=T.receiver Asset=T.asset Amount=T.amount	A certain amount (amount) of ERC-20 tokens (asset) minted from zero-address (sender) controlled by smart contract to an address (receiver)
Transfer Burn	TB	sender, receiver, amount, asset	T.receiver==zero-addres and T.amount>0	Sender=T.sender Receiver=T.receiver Asset=T.asset Amount=T.amount	A certain amount (amount) of ERC-20 token (asset) was sent to zero-address (receiver) to burn
Add Liquidity	LA	sender, receiver, pool, amount_in[],	Exists T ₁ , T ₂ and TM T ₁ .asset != T ₂ .asset	Sender=T ₁ .sender Receiver=TM.receiver	An address deposit certain amount

		asset_in[], asset_out, amount_out	T ₂ .asset != TM.asset TM.asset != T ₁ .asset	Pool = T ₁ .receiver Asset_in[0]=T ₁ .asset asset_in[1]=T ₂ .asset amount_in[0]=T ₁ .amount amount_in[1]=T ₂ .amount asset_out=TM.asset amount_out=TM.amount	(amount_in[]) of assets (asset_in[]) to pool (pool) to provide liquidity. The pool mint LP token (asset_out) to address to as a receipt.
Remove Liquidity	LR	sender, receiver, pool, amount_out[], asset_out[], asset_in	Exists T ₁ , T ₂ and TB T ₁ .asset != T ₂ .asset T ₂ .asset != TB.asset TB.asset != T ₁ .asset	sender=TB.sender receiver=T ₁ .receiver pool=T ₁ .sender asset_out[0]=T ₁ .asset asset_out[1]=T ₂ .asset amount_out[0]=T ₁ .amount amount_out[1]=T ₂ .amount asset_in=TB.asset	An address send LP token (TB.asset) back to pool to withdraw certain amount (amount_out) of token (asset_out) to remove liquidity. The LP token then send to zero-address to burn
Swap	TR	sender, receiver, amount_in, amount_out, asset_in, asset_out	Exists T ₁ , T ₂ and T ₁ .asset != T ₂ .asset and T ₁ .receiver == T ₂ .spender	sender=T ₁ .sender receiver=T ₂ .receiver pool=T ₁ .receiver asset_in=T ₁ .asset asset_out=T ₂ .asset amount_in=T ₁ .amount amount_out=T ₂ .amount	An address send a certain amount (amount_in) of assets (asset_in) to liquidity pool (pool) in exchange of certain amount (amount_out) of assets.

Stake/Deposit	ST	sender, receiver, amount_in, amount_out, asset_in, asset_out	Exists T and TM and T.receiver = TM.sender	sender=T.sender receiver=T.receiver asset_in=T.asset asset_out=TM.asset amount_in=T.amount amount_out=TM.amount	An address send a certain amount (amount_in) of assets (asset_int) to smart contract (receiver) to earn passive income.
Unstake/Withdraw	US	sender, receiver, amount_in, amount_out, asset_in, asset_out	Exists T and TB and T.receiver=TB.sender	sender=T.sender receiver=T.receiver asset_in=TB.asset asset_out=T.asset amount_in=TB.amount amount_out=T.amount	An address send receipt (asset_in) to smart contract to redeem a certain amount (amount_out) of assets(asset_out).

zero-address : 0x00

3.5 Detection Rules

We compare the result of original transaction and mutated transaction. If the returned price result is dramatically different from it in the original result, then will mark as a potentially flawed price oracle. We set the threshold for change at 40%. In DeFi space, the collateralization rate or ratio can vary significantly depending on the specific protocol and the type of collateral used. However, a common requirement is often in the range of 150% to 200%. For instance, on Aave, one of the most well-known DeFi lending platforms, the collateralization ratios vary depending on the type of asset but are typically around 75% to 80%, which corresponds to a 125% to 133% collateralization rate. The higher the collateralization rate, the more loans can be borrowed with the same value of collateral; the lower it is, the fewer loans can be made.

If a DeFi platform has set the collateralization rate of asset A at 150%, it means that if the price of A is manipulated to increase by 50%, it is possible to take out a loan that is worth more than the real value of the collateral asset. The attacker's profit is the value of the borrowed assets minus the value of the collateral. The DeFi lending platform ends up insolvent. We overwrite different state variables to simulate a mutated transaction based on different modes. For a price oracle that solely reads liquidity reserves, we exclusively modify state variables containing reserves values. In the case of a price oracle that retrieves multiple pieces of information from a liquidity pool, we

individually modify each variable to test its implementation.



Chapter 4 Experiment and Result

To eliminate network latency in our experiments on each RPC request, we set up an Ethereum archive node [53] on our local machine. It is several times faster than querying a third-party node provider such as Quicknode [54]. An archive node store historical state of Ethereum. We can query historical blockchain data and it can theoretically be built from scratch by simply replaying the blocks from genesis [55]. We choose the Erigon archive node as our execution layer rather than Geth because it has optimized its data storage format. It requires no more than 3TB of disk space. In contrast, the Geth archive node requires more than 12TB of space and at least 3 months to sync data because of its inefficiency. Since the Ethereum network has several major upgrades during our experiments, we have upgraded Erigon to version 2.42.0 to satisfy the Shapella upgrade [56]. After the merge [57], we set up Prysm and upgraded to version 4.0.0 as our consensus layer client. We use multicore CPU (AMD Ryzen 5 5600) and 64231 MB RAM on Ubuntu 22.04 LTS. Both Erigon and Prysm data are stored on a SATA 4TB SSD.

4.1 Dataset Description

DeFi Projects may uses different oracle solutions based on their requirements and resources of the project. In Figure 15, we can see that according to the Total Value

Secured (TVS) ranking, the majority of the market share is occupied by a centralized oracle, Chainlink [35], WINKLink [58], and Pyth Network [59]. Some projects use a private price oracle in their project, which is not open to the public and is maintained by its team. While using a private price oracle can offer customization and potential costs savings, it also involves centralization risk of single data sources. Other DeFi projects utilize AMM liquidity pools as its price oracle data source because the tokens they require quotes for are not mainstream, and no price oracle providers quotes for them, or for other purposes. Manipulating centralized price oracles is quite challenging and almost impossible, but inflating the price from tampering with the AMM liquidity pools is comparatively easier. Also, different projects may define different function names and may implement their price oracle in a different way. It is impossible to know every function's utility and usage in a project without its documentation and manual inspections. Most projects disclose its price oracle contract address in its white paper, but some may not. So, we select some commonly AMM liquidity pool price-related functions in Table 3 to infer the contract address and architecture of the price oracle, and use it as our dataset.

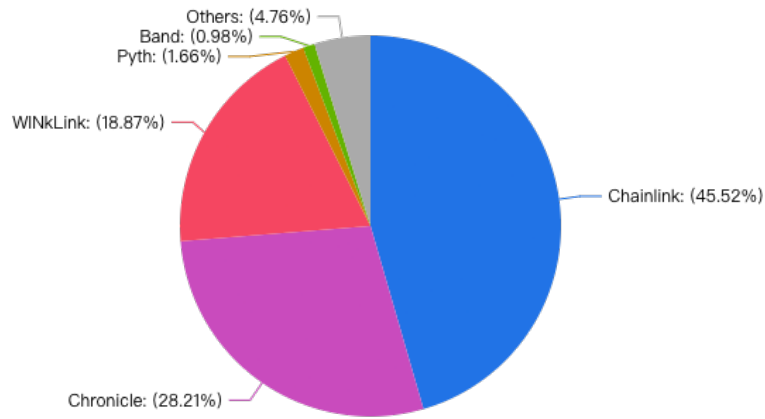


Figure 15: Market Share Pie Chart of Oracles based on TVS

Table 3: Price-related Functions

Price Data Source	Price-Related Functions	Return Value Type
ChainLink	latestAnswer() latestRoundData()	int256 (uint80,int256,uint256,uint256,uint80)
UniswapV2	getReserves() price0CumulativeLast() price1CumulativeLast()	(uint112,uint112,uint32) uint256 uint256
Sushiswap	getReserves() price0CumulativeLast() price1CumulativeLast()	(uint112,uint112,uint32) uint256 uint256
Curve.StableSwap	get_virtual_price() get_dy()	uint256 uint256
Curve.MetaPool	get_price_cumulative_last() get_twap_balances() get_dy()	uint256[] uint256[] uint256

We collected transactions which contains the price-related function, *getReserves()*, in its invocation flow from different time periods, as shown in Table 4 and Table 6. We

also collect from the same time period as the ProMutator’s dataset as shown in Table 5

as a comparison. A transaction may contain more than one of our targeted function calls.

Table 4: Dataset Description at Different Timestamp

Dataset Tag	Start Block	End Block	Transaction(s)	Function Call(s)
Dataset_1100	11,000,000	11,009,999	285,504	400,047
Dataset_1200	12,000,000	12,099,999	258,902	341,896
Dataset_1400	14,000,000	14,099,999	77,809	104,762
Dataset_1500	15,000,000	15,099,999	74,475	104,388
Dataset_1600	16,000,000	16,099,999	90,469	114,758

Table 5: Dataset Description Compared to ProMutator

Dataset Tag	Start Block	End Block	Transaction(s)	Function Call(s)
Dataset_1109	11,090,000	11,099,999	231,348	310,907
Dataset_1110	11,100,000	11,109,999	223,888	298,049
Dataset_1111	11,110,000	11,119,999	205,708	280,528
Dataset_1112	11,120,000	11,129,999	211,496	286,826
Dataset_1113	11,130,000	11,139,999	226,355	306,446
Dataset_1114	11,140,000	11,149,999	260,656	351,722
Dataset_1115	11,150,000	11,159,999	283,378	379,586
Dataset_1116	11,160,000	11,169,999	238,615	320,030
Dataset_1117	11,170,000	11,179,999	253,719	331,487
Dataset_1118	11,180,000	11,189,999	208,220	285,435
Dataset_1119	11,190,000	11,199,999	221,569	304,038
Dataset_1120	11,200,000	11,209,999	196,930	263,604
Dataset_1121	11,210,000	11,219,999	193,629	263,420
Dataset_1122	11,220,000	11,229,999	209,035	289,360
Dataset_1123	11,230,000	11,239,999	199,120	276,768
Dataset_1124	11,240,000	11,249,999	190,886	258,846
Dataset_1125	11,250,000	11,259,999	209,391	289,082
Dataset_1126	11,260,000	11,269,999	223,161	308,876
Dataset_1127	11,270,000	11,279,999	230,429	322,098
Dataset_1128	11,280,000	11,289,999	201,375	275,518

Dataset_1129	11,290,000	11,299,999	232,499	312,118
Dataset_1130	11,300,000	11,309,999	238,381	323,422
Dataset_1131	11,310,000	11,319,999	237,357	313,191
Dataset_1132	11,320,000	11,329,999	214,531	293,613
Dataset_1133	11,330,000	11,339,999	193,166	269,601
Dataset_1134	11,340,000	11,349,999	193,330	267,044
Dataset_1135	11,350,000	11,359,999	191,467	268,880
Dataset_1136	11,360,000	11,369,999	198,608	274,425
Dataset_1137	11,370,000	11,379,999	229,009	324,095
Dataset_1138	11,380,000	11,389,999	263,772	365,385
Dataset_1139	11,390,000	11,399,999	259,568	360,632
Dataset_1140	11,400,000	11,409,999	317,447	440,999
Dataset_1141	11,410,000	11,419,999	298,494	433,808
Dataset_1142	11,420,000	11,429,999	304,542	436,987
Dataset_1143	11,430,000	11,439,999	260,637	362,808
Dataset_1144	11,440,000	11,449,999	239,335	348,571
Dataset_1145	11,450,000	11,459,999	256,417	365,746
Dataset_1146	11,460,000	11,469,999	210,978	289,537
Dataset_1147	11,470,000	11,479,999	228,166	332,792
Dataset_1148	11,480,000	11,489,999	248,905	363,309

Table 6: Dataset Description of Recent Transactions

Dataset_1700	17,000,000	17,009,999	93,268	112,573
Dataste_1701	17,010,000	17,019,999	103,441	122,468
Dataset_1702	17,020,000	17,029,999	107,281	131,189
Dataset_1703	17,030,000	17,039,999	104,248	128,263
Dataset_1704	17,040,000	17,049,999	119,878	149,327
Dataset_1705	17,050,000	17,059,999	100,314	122,152
Dataset_1706	17,060,000	17,069,999	223,216	267,775
Dataset_1707	17,070,000	17,079,999	296,036	338,317
Dataset_1708	17,080,000	17,089,999	324,368	366,634
Dataset_1709	17,090,000	17,099,999	213,474	264,927

4.2 Baseline Methods

To evaluate the credibility of our price oracle vulnerabilities detection, we run our experiments on known attacks. We select attacks that happened on Ethereum mainnet from DefiLlama [1], in which the root cause of the attack was related to price oracle in Table 7. While some attacks can be detected by our methods, others cannot. Take the Euler Finance attack for example, where the attackers utilize the flawed donate function to liquidate itself. The price oracle function works fine in this case; we are not able to detect it using our proposed method. Our criteria for judging "Yes" are that the root cause of the price oracle attack can be found by our method with normal transactions as its input.

Table 7: Attacks Related to Price Oracle on Ethereum Mainnet

Project Name	Date	Technique	Possible to Detect
Euler Finance	13 Mar, 2023	Flashloan Donate Function Logic Exploit	No
Elastic Swap	13 Dec, 2022	Price Oracle Attack	No
Inverse Finance 2 nd	16 Jun, 2022	Flashloan Price Oracle Attack	Yes
Inverse Finance 1 st	2 Apr, 2022	Price Oracle Attack	Yes
Cream 3 rd	27 Oct, 2021	Flashloan Price Oracle Attack	No
Indexed Finance	14 Oct, 2021	Flashloan Price Oracle Attack	Yes
Growth DeFi	9 Feb, 2021	Flashloan Price Oracle Attack	No
Yearn	5 Feb, 2021	Flashloan Price Oracle Attack	No
Warp Finance	18 Dec, 2020	Flashloan Price Oracle Attack	Yes
CheeseBank	16 Nov, 2020	Flashloan Price Oracle Attack	Yes
Value DeFi	14 Nov, 2020	Flashloan Price Oracle Attack	No
Harvest Finance	26 Oct, 2020	Flashloan Price Oracle Attack	Yes
Eminence	28 Sep, 2020	Flashloan Price Oracle Attack	No
bZx 2 nd	18 Feb, 2020	Flashloan Price Oracle Attack	No

4.3 Comparison Criteria

We set the threshold of percentage change of mutated price and the origin price at 40%. If a price can be manipulated higher or lower than the threshold, then we will count it as a flawed price oracle. The reason being that we do not require the manipulated test results to be exactly the same as the original transaction to prove it is immune to malicious behavior. The returned result of simulating a function call using `eth_call` is not always identical to its internal function call returned result, because the `eth_call` simulates the message call at the end of the block, which means the position of the origin transaction in the block is almost always different. So, we set a threshold, rather than check if it is equal.

If the test result does not exceed the threshold, it indicates that the price oracle has passed and the price has not been manipulated; otherwise, it is a defect. There is another possibility that the `eth_call` can be reverted. Transactions can be reverted or fail for various reasons, which means that the intended changes to the blockchain state were not applied, and the transaction did not successfully complete. Meanwhile, if `eth_call` is reverted, it typically means that certain conditions are not met that trigger a revert. It is a sanity check to prevent unintended behavior or protect the smart contract from potential security vulnerabilities.

Afterwards, we manually review the potential flawed price oracles that we marked

in our experimentation. We take the price oracle 's function name, its function call graph, and transaction behavior into consideration to determine whether it is a true positive (TP) or a false positive (FP).

4.4 Experimental Results

In this chapter, we will discuss the experimental results and how we do the manual inspections.

4.4.1 Output of Transaction Simulation

In Table 8 to Table 10, we demonstrate the results of our simulation results on price oracle smart contracts. There are three outcomes we will get from a simulation test: Defect, Revert, and Passed. *Defect* means that the mutated-value override in the price oracle simulation test has changed the price from the original price over the threshold. This means that there is the potential that skewing the liquidity pool can easily manipulate the price. *Passed* means that the mutated state variables don't change the price or the simulated price doesn't go higher or lower than the threshold. A simulation using message call may not get the same result from the original transaction, including its internal function results. Message call simulate the transaction or the function calls at the end of the block, while the origin transactional may be at the middle index of the

block. Other transactions in the same block that occur after the transaction that we are testing may change variables related to the price oracle. As a result, the obtained results may differ. So, we allow for the slight differences between the tested results and the original results. *Revert* means “transaction revert” in short, typically due to an error in smart contract execution or some condition specified in the contract was not met. In our test, reverts usually occur when we modify variables to the extent that they no longer satisfy the sanity check of the price oracle in the original code. For example, when the price oracle is calculating the asset price, it refers to the from multiple liquidity pool or other aggregated price oracles. If the price of one of them deviated too much from the others, it will be reverted. It is a good and reasonable design for a price oracle. However, we cannot determine if the price oracle is truly flawless just based on the revert results alone.

4.4.2 Results of Detection and Manual Inspections

In Tables 8 to 9, we found that the price oracles detected as having potential vulnerabilities often turn out to be the same one, appearing repeatedly. For example, in Dataset_1500, we detect 2527 potential vulnerabilities, but 2495 of them are the same smart contract and are determined to be false positives after manual analysis. In some cases, the smart contracts that our method marks as vulnerable are not necessarily used

as price oracles for calculating prices for other smart contracts. Some transactions might merely calculate the number of tokens in liquidity pools, which does not result in losses. Figure 16 is an example of a transaction [60] transfer token, while the price oracle only calculates its transfer value. It was marked vulnerable by our method. However, it is a closed-source smart contract, which means we can't get the source code. But we can guess it is a vulnerable price oracle based on its invocation flow. It only accesses the liquidity pool reserves to calculate the price rather than calling `price0CumulativeLast()` or `price1CumulativeLast()`.

We make a comprehensive judgement as to whether the price oracle is vulnerable or not by each transaction information we get. We may get an opposite result of the same price oracle because of different information we get. For example, if a price oracle calculates the output assets amount using the input assets amount as the parameter to get the ratio of the liquidity pool assets to calculate price, then it is a true positive, because the result from the flawed price oracle is used as quoting assets.

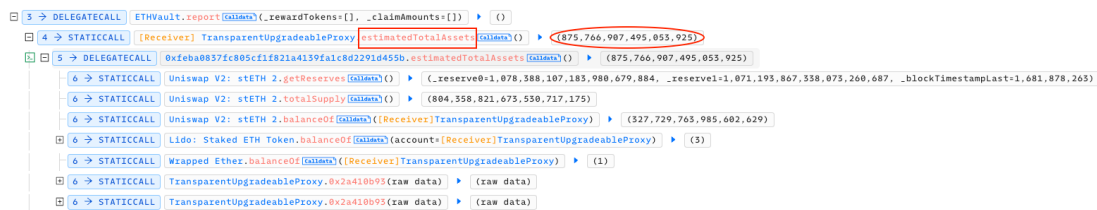


Figure 16: False Positive of Detecting Price Oracle Vulnerability

There is another situation where an arbitrageur user used the same function as calculating the output amount. The arbitrageur calculates its arbitrage route and amounts off-chain, so the output amount it gets from the price oracle only used to check its minimum received from the liquidity pool. We judge the price oracle to be a false positive in this situation. In addition, our method also found some rebase token misuse liquidity pool reserves to calculate its shares in its rebase mechanism. We also prove that our method is available to run through a transaction that contains a non-verified smart contract, which means the deployer of the smart contract does not reveal its smart contract source code. In Figure 17, it is obvious that the price oracle [61] uses the asset ratio in the ETH-USDT liquidity pool as the price in base of ether (ETH), which is then multiplied by the transfer amount to get the token transfer values. The implementation can be simplified to Equation 5.

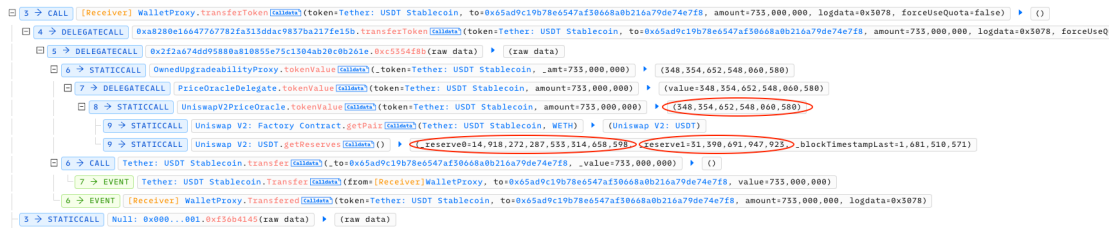


Figure 17: Harmless Price Oracle Vulnerability

$$V = a \times \frac{r_0}{r_1} \quad (5)$$

The time spent on each test by our method is largely dependent on the performance of the machine that we had set up in our archive node. The simulation of message call is carried out in the local virtual machine within the node. Compared to the full

transaction simulation services of invocation flows provided to developers by other third-party companies, such as Tenderly and Alchemy, our method requires significantly less time for each transaction. Therefore, it can be applied for real-time monitoring or early warning systems.

During our manual inspection of true positives and false positives, we found that we cannot determine whether a result is a true positive or false positive based solely on the contents of each transaction, including function names, called functions, etc. In fact, we may get different results from different transactions, even with the same price oracle smart contract, due to different actions performed within the transactions.

Comparing Table 9 and Table 10, we can find that the average potential vulnerabilities per ten thousand blocks have decreased dramatically. We believe that developers are becoming increasingly aware that this type of price calculation method is susceptible to manipulation attacks. With the growing popularity of UniswapV3, when project teams need to use liquidity pools to calculate a price, they gradually tend to combine the prices obtained from both UniswapV2 and UniswapV3.

Table 8: Simulation Results at Different Timestamp

Dataset Tag	Function Calls	Pass	Revert	Defect	TP	FP
Dataset_1100	400,047	101,698	298,341	8	4	4
Dataset_1200	341,896	87,484	254,249	172	99	73
Dataset_1400	104,762	14,254	90,474	34	19	15
Dataset_1500	104,388	24,707	77,154	2527	9	2518
Dataset_1600	114,758	28,398	86,927	63	39	24

Table 9: Simulation Results Compared to ProMutator

Dataset Tag	Function Calls	Pass	Revert	Defect	TP	FP
Dataset_1109	310,907	90,290	220,544	73	9	64
Dataset_1110	298,049	92,063	205,881	105	12	93
Dataset_1111	280,528	95,542	187,739	237	3	234
Dataset_1112	286,826	95,557	191,067	202	89	113
Dataset_1113	306,446	101,980	203,920	546	286	260
Dataset_1114	351,722	107,979	243,566	177	172	5
Dataset_1115	379,586	112,759	266,617	210	112	98
Dataset_1116	320,030	102,016	217,036	978	57	921
Dataset_1117	331,487	78,147	252,736	604	22	582
Dataset_1118	285,435	68,658	215,769	1008	94	914
Dataset_1119	304,038	125,976	177,635	427	21	406
Dataset_1120	263,604	63,164	199,480	960	31	929
Dataset_1121	263,420	67,216	196,175	29	29	0
Dataset_1122	289,360	93,441	195,891	28	28	0
Dataset_1123	276,768	8,758	190,311	51	48	3
Dataset_1124	258,846	33,727	157,089	70	37	30
Dataset_1125	289,082	11,965	197,333	93	27	66
Dataset_1126	308,876	130,536	178,305	35	11	24
Dataset_1127	322,098	99,028	222,976	94	68	26
Dataset_1128	275,518	78,927	196,421	170	12	158
Dataset_1129	312,118	133,202	178,791	125	43	82
Dataset_1130	323,422	90,658	232,594	170	94	76
Dataset_1131	313,191	81,579	231,435	177	27	150
Dataset_1132	293,613	124,361	169,145	107	36	71
Dataset_1133	269,601	144,779	124,637	185	177	8
Dataset_1134	267,044	94,852	171,243	949	397	552
Dataset_1135	268,880	87,699	180,402	779	645	134
Dataset_1136	274,425	78,548	195,482	395	109	286
Dataset_1137	324,095	112,703	210,904	488	50	438
Dataset_1138	365,385	111,264	253,585	536	114	422
Dataset_1139	360,632	164,691	195,534	407	60	347
Dataset_1140	440,999	139,100	301,000	899	90	809
Dataset_1141	433,808	134,600	298,620	588	94	494
Dataset_1142	436,987	192,114	244,481	392	77	315

Dataset_1143	362,808	197,948	164,686	174	33	141
Dataset_1144	348,571	144,113	203,978	480	35	445
Dataset_1145	365,746	123,114	242,277	355	83	272
Dataset_1146	289,537	88,251	200,500	786	415	371
Dataset_1147	332,792	99,028	222,976	523	296	227
Dataset_1148	363,309	78,927	196,421	267	88	179

Table 10: Simulation Results of Recent Transactions

Dataset Tag	Function Calls	Pass	Revert	Defect	TP	FP
Dataset_1700	112,573	87,391	25,166	16	15	1
Dataset_1701	122,468	93,453	29,010	5	2	3
Dataset_1702	131,189	102,169	29,004	16	3	13
Dataset_1703	128,263	100,685	27,556	22	4	18
Dataset_1704	149,327	115,331	33,957	39	1	38
Dataset_1705	122,152	95,518	26,557	77	70	7
Dataset_1706	267,775	186,287	81,446	42	33	9
Dataset_1707	338,317	223,518	114,787	12	8	4
Dataset_1708	366,634	241,056	125,562	16	4	12
Dataset_1709	264,927	190,747	74,155	25	16	9

Figure 18 shows the occurrences of the potentially flawed price oracle our method detects. Some of them appear multiple times across multiple datasets. There is a total of 103 unique potential price oracle smart contracts that were found by us. 38 of them are true positive, which is vulnerable to its user. 65 of them are false positives inspected by us manually.

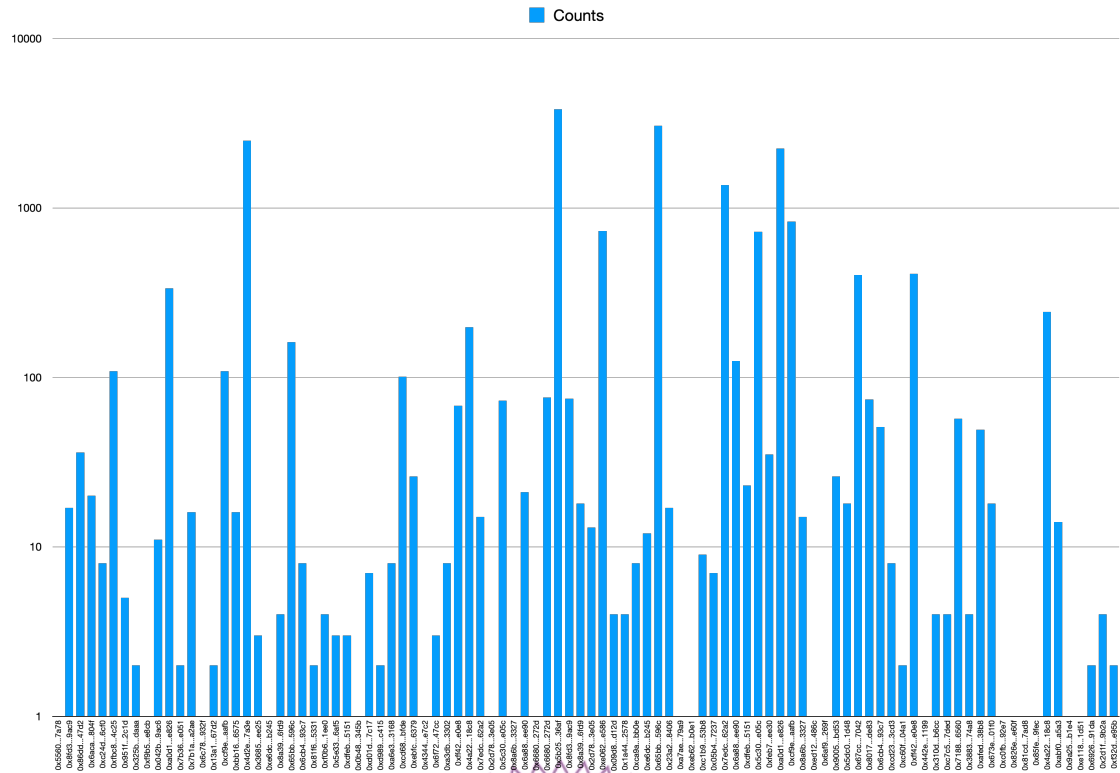


Figure 18: Occurrences Chart of Detected Potential Price Oracle



Chapter 5 Threats to Validity and Research Questions

5.1 Threats to Validity

The degree of validity in the experimental design is just as crucial as the obtained results. When the validity is sufficient, it signifies that the study's findings have a strong capacity to address analogous issues. In the subsequent subsections, we will provide a concise examination of the internal, external, construct and conclusion of the experiment, respectively.

5.1.1 Internal Validity

Internal Validity refers to the extent to which a research design can establish a causal relationship between the independent variable and the dependent factors in that different logic designs and various implementation ways use price oracles in DeFi projects. This prevents our experiment from presenting a highly effective universal solution for detecting price oracle vulnerabilities. If we make our experimental design too simplistic, we might easily overlook the true vulnerabilities. However, if the design is too complicated, it could lead to a high false positive rate, which means we need to spend extra effort to review these errors. To reduce this threat to validity, we only mutate balances in the liquidity pool and call the price-related function to get the affected results. Then, we simulate the origin transaction with the mutate balances to get the

receipt, showing whether the transaction had been successful or had failed.

5.1.2 External Validity

External validity refers to the extent to which the results of a study can be generalized or applied to other situations. We will discuss some of the external validity of our study in this subsection. Current analysis tools emphasize identifying code errors in smart contracts, and to our best knowledge, there is virtually no way to uncover logic errors. To improve the generalizability, we replay the transaction and trace the involved storage slots in each contract. By recording the involved storage slot, we can overwrite state variables to test each internal function in the transaction. Although it is possible to test other logic errors by overwriting state variables, detection rules may differ in different situations. Research still needs to be conducted with more experiments and comparisons in the future.

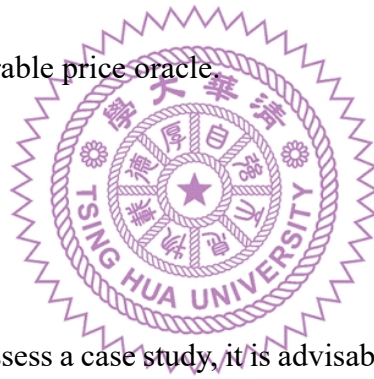
5.1.3 Construct Validity

Construct validity asks whether or not the correct factors have been leveraged to illustrate our conclusions. To verify that our proposed method can detect vulnerable price oracles, we apply our approach to past attacks in Table 7 and Chapter 3. Furthermore, we compare our results with the ProMutator [17] within the same block

interval.

5.1.4 Conclusion Validity

Conclusion validity describes whether the conclusions we get from our experiment results are reasonable. How we create the dataset from real-world transactions is a primary threat to conclusion validity. However, there isn't an existing dataset that is already organized and annotated. We evaluate our approach with different block period datasets to test different kinds of price oracle and design patterns to minimize the effect of keep getting same vulnerable price oracle.



5.2 Research Questions

To comprehensively assess a case study, it is advisable to address specific research questions. In this section, we will concisely present the experimental results that provide answers to four key research questions.

RQ1: How does the simulation-based detection method detect the price oracle which is vulnerable to market manipulation by comparing two different results?

We parse real transactions to get the relationship between price oracle and liquidity

pool. Then we mutate the liquidity pool smart contract state variables to inflate the price at the price oracle. The simulation result of a flawlessly functional price oracle should remain identical to the original result. If the result can be changed then we can infer that the price oracle is not immune to market manipulation. The threshold we set at our experiment is to discriminate which price oracle will change its result dramatically. The variables we override are multiple times greater than its original values to simulate extreme market manipulation. Therefore, if the simulation result of a price oracle changes, it will change dramatically. Obviously, we can tell which price oracle is vulnerable to market manipulation by its variation of origin and simulation results.


RQ2: Will different time range and price-related functions which we select in our dataset and simulation scenario and which we design affect the coverage and performance of our simulation-based detection method?

Yes, our experimental result is mainly based on the transaction we filtered out from the blockchain and the different attack vectors we simulated. We only choose multiple mainstream AMM protocols and may have overlooked some other AMM protocols which has different function signatures, though with only slightly modifications. We learned from past attack incidents to design our simulation scenario, which may not be possible to prevent brand new attack vectors. But we can detect the price oracle which

may be affected by the same attacks that happened in the past.

RQ3: Why does our experiment exclude the centralized price oracle?

We assume that a project's price oracle may use manipulated data to calculate a price in our experiments. However, the centralized price oracle set up an oracle node network and then aggregates all price data which is published by each node. If a node publishes outlier data, the aggregator algorithm will not accept and discard it ideally. It is impractical to assume that the majority of oracle nodes all publish inflated price data, or the aggregator algorithm will fail. Price manipulation attacks usually occurs in a project that uses AMM protocol as its price data feed. Focusing on finding an AMM-based vulnerable price oracle is a better way to improve the safety of DeFi.



RQ4: What is the benefit of our simulation-based detection method?

Our method is based on message call, using `eth_call` on the RPC node to simulate a price manipulation attack. It is faster than forking the blockchain on local environment and writing test files on Foundry and truffle if we only want to check whether the price oracle is vulnerable or not. Also, our simulation scenario is scalable and available on other EVM compatible chains. The user can design their attack vector to simulate different scenarios on their own.

Chapter 6 Conclusion and Future Work

The struggle between auditors and hackers is akin to a clash between the spear and the shield. Both sides will continue to make progress. In this study, we focus on finding and detecting price oracle manipulation weaknesses. We proposed a novel method on detecting a vulnerable price oracle, which skewed the liquidity pool reserves to simulate a manipulation attack. Our method doesn't need to know how the price oracle is designed, and to which liquidity pool the price oracle referred to beforehand. It parsed the smart contract relationship in the transaction to get the price oracle information. It is useful for risk-taking users, who are adventurous to new DeFi projects, to take fast inspections on the project's smart contract safety.

Our method is also available for testing close-sourced price oracle smart contracts, which is a significant advantage over other tools. We utilize the input parameters and function calls in the transaction details to analyze the closed-sourced price oracle designs, even though we haven't completed many attack vector simulations yet. We still find 38 unique price oracles which are vulnerable to price manipulation, and some of them are closed-sourced. Our method has also proven that it can detect some vulnerable price oracles that have been previously attacked, allowing the developers an early prevention and remediation capability.

Since there is a limitation in using the message call on the smart contract, we run

one simulation attack vector for each piece of test transaction at a time. The results of different simulation attack vectors independently determine whether the price oracle is vulnerable or not. Therefore, we will manually review the same price oracle multiple times, and we hope to combine different results together in the future. In the manual review process, the accuracy of judgment relies heavily on the reviewer's relevant knowledge background and the richness of the data.

In our future work, we are attempting to include more types of liquidity pools: Uniswap V3 and Uniswap V4. The Uniswap V4 draft was published in June 2023, as well as adding more simulation scenarios and more types of liquidity pools to emulate a wider range of attack vectors. We are also attempting to combine multiple simulation attack vector in one test transaction, which allows us to get closer to reality and analyze the impact on the project's smart contract. In addition, we did not take into account the impact of liquidity depth on price changes in our mutated variables, while liquidity pool depth is also another important factor that may affect the price in the liquidity pool. Last but not least, our simulation method can only get the result of the transaction or the function call, so we decide to add our customized RPC node method to both support simulation and record parameters in each invocation flow, which will make our test circle more efficient and time-saving.

References

- [1] "DefiLlama, " <https://defillama.com/> (accessed Mar. 2023).
- [2] "Aku, " <https://aku.world/> (accessed Mar. 2023).
- [3] Fortune, "How an NFT creator lost \$34 million due to a smart contract error, " <https://fortune.com/2022/04/26/micah-johnson-akutars-nft-launch-smart-contracts-error/> (accessed July. 2023)
- [4] J. Feist, G. Grieco and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts." *In Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, Montreal, Canada, 2019, pp. 8-15, doi:10.1109/WETSEB.2019.00008.
- [5] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts." *In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, New York, USA, 2018, pp. 9-16, doi:10.1145/3194113.3194115.
- [6] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts." *In Proceedings of the 25th Network and Distributed System Security Symposium Internet Society (NDSS)*, San Diego, USA, 2018, doi:10.14722/ndss.2018.23092.

- [7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts." *In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, New York, NY, USA, 2018, pp. 67–82, doi:10.1145/3243734.3243780.
- [8] Quantstamp, "What is a Re-entrancy Attack, " <https://quantstamp.com/blog/what-is-a-re-entrancy-attack> (accessed Mar. 2023).
- [9] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit." *In Proceedings of the 9th Hack in The Box Security Conference (HITB SECCONF)*, Amsterdam, Netherland, 2018, vol. 9:54.
- [10] G. Grieco, Will Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts." *In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, New York, USA, 2020, pp. 557–560, doi:10.1145/3395363.3404366.
- [11] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts." *In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, USA, 2019, pp. 1186-1189, doi:10.1109/ASE.2019.00133.
- [12] Chainalysis, "Oracle Manipulation Attack Rising: A Unique Concern for DeFi, "

<https://blog.chainalysis.com/reports/oracle-manipulation-attacks-rising/>

(accessed Apr. 2023).

- [13] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities." *In Proceedings of IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084-2106, 1 Oct. 2021, doi: 10.1109/TSE.2019.2942301.
- [14] Euler, "EToken Smart Contract, " <https://etherscan.io/address/0xbb0d4bb654a21054af95456a3b29c63e8d1f4c0a#code> (accessed Apr. 2023).
- [15] Sherlock, "Euler Finance Audit Report," July 2022, https://www.hacknote.co/17c261f7d8fWbdml/1821f966f40pJG_1 (accessed Apr. 2023)
- [16] W. Li, J. Bu, X. Li and X. Chen, "Security Analysis of DeFi: Vulnerabilities, Attacks and Advances." *In Proceedings of the 4th IEEE International Conference on Blockchain (ICBC)*, Espoo, Finland, 2022, pp. 488-493, doi: 10.1109/Blockchain55522.2022.00075.
- [17] S. H. Wang, C. C. Wu, Y. C. Liang, L. H. Hsieh and H. C. Hsiao, "ProMutator: Detecting Vulnerable Price Oracles in DeFi by Mutated Transactions." *In Proceedings of the 6th IEEE European Symposium on Security and Privacy Workshops(EuroS&PW)*, Vienna, Austria, 2021, pp. 380-385, doi: 10.1109/EuroSPW54576.2021.00047.

- [18] B. Wang, H. Lin, C. Liu, Z. Yang, Q. Ren, H. Zheng and H. Lei, "BLOCKEYE: Hunting for DeFi Attacks on Blockchain." *In Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Madrid, Spain, 2021, pp. 17-20, doi:10.1109/ICSE-Companion52605.2021.00025.
- [19] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Qi. He and K. Ren, "DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications." arXiv preprint arXiv:2104.15068 (2021).
- [20] Investopedia, "Market Maker Definition: What It Means and How They Make Money, " <https://www.investopedia.com/terms/m/marketmaker.asp> (accessed Mar. 2023).
- [21] CoinDesk, "What Is an Automated Market Maker, " <https://www.coindesk.com/learn/what-is-an-automated-market-maker/> (accessed Mar. 2023).
- [22] Vitalik Buterin, "On Path Independence, " <https://vitalik.ca/general/2017/06/22/marketmakers.html> (accessed Mar. 2023).
- [23] Uniswap Lab, "Uniswap V1 whitepaper, " <https://hackmd.io/@HaydenAdams/HJ9jLsfTz> (accessed Mar. 2023).
- [24] Uniswap Lab, "Uniswap V2 whitepaper, " <https://uniswap.org/whitepaper.pdf> (accessed Mar. 2023).

- [25] Ethereum Foundation, "Oracles, " <https://ethereum.org/en/developers/docs/oracles/> (accessed July 2023).
- [26] DefiLlama, "The oracle problem, " https://wiki.defillama.com/wiki/The_oracle_problem (accessed July 2023).
- [27] G. Caldarelli, "Understanding the Blockchain Oracle Problem: A Call for Action." *In Proceedings of Information*, vol. 11, no. 11:509, Nov. 2020.
doi:10.3390/info11110509
- [28] Curran, B., "What are Oracles? Smart Contracts, Chainlink & The Oracle Problem, " <https://blocknomi.com/oracles-guide> (accessed July 2023).
- [29] Pyth Network, "Pyth Root Cause Analysis, " <https://pythnetwork.medium.com/pyth-root-cause-analysis-622376d7a492> (accessed July 2023).
- [30] Samczsun, "So you want to use a price oracle," <https://samczsun.com/so-you-want-to-use-a-price-oracle/> (accessed Apr. 2023).
- [31] Max Wolff, "Introducing Marble, " <https://medium.com/marbleorg/introducing-marble-a-smart-contract-bank-c9c438a12890> (accessed July 2023).
- [32] S. Eskandari, M. Salehi, W. C. Gu, and J. Clark, "SoK: Oracles from the Ground Truth to Market Manipulation." *In Proceedings of the 3rd ACM Conference on Advances in Financial Technologies (AFT)*, Arlington, USA., pp. 127-141,
doi:10.1145/3479722.3480994.

- [33] Chainalysis, "\$197 Million Stolen: Euler Finance Flash Loan Attack Explained, "
<https://blog.chainalysis.com/reports/euler-finance-flash-loan-attack/> (accessed
July 2023).
- [34] Ethereum, "Execution API Specification", [https://github.com/ethereum/execution](https://github.com/ethereum/execution-api)
[-apis](https://github.com/ethereum/execution-api) (accessed Apr. 2023)
- [35] Chainlink, <https://chain.link/> (accessed Apr. 2023).
- [36] Cmicel, "Replaying Ethereum Hacks – Rari Fuse VUSD Price Manipulation, "
<https://cmichel.io/replaying-ethereum-hacks-rari-fuse-vusd-price-manipulation/>
(accessed June 2023).
- [37] PeckShield, "Kill Switch, "<https://alert.peckshield.com/> (accessed Apr. 2023).
- [38] Solidity, "Using the Compiler, "[https://docs.soliditylang.org/en/latest/using-the-](https://docs.soliditylang.org/en/latest/using-the-compiler.html)
[compiler.html](https://docs.soliditylang.org/en/latest/using-the-compiler.html) (accessed June 2023).
- [39] Ethereum, "JSON-RPC Server, "[https://geth.ethereum.org/docs/interacting-with-](https://geth.ethereum.org/docs/interacting-with-geth/rpc)
[geth/rpc](https://geth.ethereum.org/docs/interacting-with-geth/rpc) (accessed Apr. 2023).
- [40] Erigon, "RPC Implementation Status," [https://github.com/ledgerwatch/erigon/](https://github.com/ledgerwatch/erigon/blob/devel/cmd/rpcdaemon/README.md#rpc-implementation-status)
[blob/devel/cmd/rpcdaemon/README.md#rpc-implementation-status](https://github.com/ledgerwatch/erigon/blob/devel/cmd/rpcdaemon/README.md#rpc-implementation-status) (accessed
Apr. 2023).
- [41] Alchemy, "debug_traceTransaction," [https://docs.alchemy.com/reference/debug-](https://docs.alchemy.com/reference/debug-tracetransaction)
[tracetransaction](https://docs.alchemy.com/reference/debug-tracetransaction) (accessed Apr. 2023).

- [42] Alchemy, "trace_transaction," <https://docs.alchemy.com/reference/trace-transaction> (accessed Apr. 2023).
- [43] Chainstacklabs, "Debug transactions using Erigon and Geth," <https://github.com/chainstacklabs/debug-trace-transactions-erigon/blob/main/README.md#debug-transactions-using-erigon-and-geth> (accessed Apr. 2023).
- [44] Ethereum, "Custom EVM Tracer," <https://geth.ethereum.org/docs/developers/evm-tracing/custom-tracer> (accessed Apr. 2023).
- [45] Etherscan, "Transaction Details," <https://etherscan.io/tx/0x76646543e8b079e9095b46fbf853ec79afa3335fccf5c1e9792460b0523eeca> (accessed Apr. 2023).
- [46] Tenderly Explorer, <https://dashboard.tenderly.co/explorer> (accessed Apr. 2023).
- [47] Etherscan, "Keep3rV2Oracle Smart Contract," <https://etherscan.io/address/0x39b1dF026010b5aEA781f90542EE19E900F2Db15#code> (accessed May 2023).
- [48] Solidity, "Layout of State Variables in Storage," https://docs.soliditylang.org/en/v0.8.19/internals/layout_in_storage.html
- [49] Uniswap, "ExampleOracleSimple Smart Contract", <https://github.com/Uniswap/v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol> (accessed Apr. 2023).
- [50] Etherscan, "Transaction Details," <https://etherscan.io/tx/0x920f87a1338e0a4ddc>

[f382e9c9cd4e6bc68e3f6d22f3c6efce31e122d08b3a2c](#) (accessed Apr. 2023).

[51] Etherscan, "Zero-Value Token Transfer Attack, " <https://info.etherscan.com/zero-value-token-transfer-attack/> (accessed May 2023).

[52] P. Xia, H. wang, B. Gao, W. Su, Z. Yu, X. Luo, C. Zhang, X. Xiao, G. Xu, "Trade or Tick? Detecting and Characterizing Scam Tokens on Uniswap Decentralized Exchange." *In Proceedings of the ACM on Measurement and Analysis on Computer Systems*, vol. 5, no. 3, pp 1-39, Dec. 2021, doi:10.1145/3491051.

[53] Ethereum Foundation, "Ethereum Archive Node, " <https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes/> (accessed Apr. 2023).

[54] Quicknode, <https://quicknode.com/> (accessed Apr. 2023).

[55] Quicknode, "Ethereum Full Node vs. Archive Node, " <https://www.quicknode.com/guides/infrastructure/node-setup/ethereum-full-node-vs-archive-node/> (accessed Apr. 2023).

[56] Ethereum Foundation, "Mainnet Shapella Announcement, " <https://blog.ethereum.org/2023/03/28/shapella-mainnet-announcement> (accessed Apr. 2023).

[57] Ethereum Foundation, "The Merge, " <https://ethereum.org/en/roadmap/merge/> (accessed Apr. 2023).

[58] WINkLink, <https://winklink.org/> (accessed Apr. 2023).

- [59] Pyth Network, <https://pyth.network/> (accessed Apr. 2023).
- [60] Blocksec, "Phalcon Explorer, " <https://explorer.phalcon.xyz/tx/eth/0xea7b0aa6e50c222fc6d550d1bf2142a285e74a336dc7258970246257f7adfc34> (accessed June 2023).
- [61] Blocksec, " Phalcon Explorer, " <https://explorer.phalcon.xyz/tx/eth/0x999a984d0e7462d5fef80614b5ec857acf3b3ef3e6ae3d590fb6c0295781014c> (accessed June 2023).

