ProMutator: Detecting Vulnerable Price Oracles in DeFi by Mutated Transactions

Shih-Hung Wang, Chia-Chien Wu, Yu-Chuan Liang, Li-Hsun Hsieh and Hsu-Chun Hsiao
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
{*b04902019, r09922191, r08922008, r09922185, hcshiao*}*@csie.ntu.edu.tw*

***Abstract*—This paper presents ProMutator, a scalable security analysis framework that detects price oracle vulnerabilities *before* attacks occur. ProMutator's core idea is to simulate price oracle attacks locally by mutating the data needed for price calculation. ProMutator analyzes existing transactions to reconstruct probable DeFi use patterns, thereby reducing the required simulation runs drastically. ProMutator does not require any examined contracts' high-level source code. Additionally, ProMutator generates a report for each detected vulnerability to facilitate further investigation. In our evaluation, ProMutator successfully discovered five out of six known and 27 new price oracle vulnerabilities in DeFi protocols.**

***Index Terms*—DeFi, price oracle, price oracle attack**

1. Introduction

Decentralized Finance (DeFi) allows users to trade financial products on a distributed system, typically a blockchain, thus eliminating the dependency on centralized brokers (e.g., banks). Since its debut in 2018, DeFi's total value locked (TVL) has rapidly grown to around 60 billion US dollars [1]. This high TVL has attracted attackers exploiting *DeFi composability* [2], [3] for financial gains. DeFi composability allows developers to build complex financial services by combining DeFi protocols, creating a deeply interconnected ecosystem. Consequently, a vulnerable smart contract or DeFi protocol can indirectly affect many others depending on it.

A notable example is *price oracle attacks*, whose cost exceeded 43 million dollars in 2020. Price oracles provide price information of digital assets to a wide variety of DeFi protocols and are critical components in the DeFi ecosystem. In a price oracle attack, the attacker manipulates the price information to profit from victims relying on it. Unfortunately, existing security tools are either 1) limited to conducting post-incident analysis [4]–[9] or 2) focusing on security issues within individual smart contracts and may scale poorly when being applied to composed DeFi protocols [10], [11].

This work presents ProMutator, a scalable security analysis framework that discovers price oracle vulnerabilities *before* attacks occur, thereby protecting DeFi developers from devastating financial loss. ProMutator takes advantage of existing transactions as references to learn how DeFi protocols are composed with others, and mutates the transactions to proactively find potential vulnerabilities instead of reactively reporting the attack incidents. Moreover, ProMutator reports high-level attack steps for further manual verification. Our preliminary evaluation shows promising results. ProMutator successfully re-discovered five known attacks using transactions predating these attacks and found 27 confirmed price oracle vulnerabilities in existing DeFi protocols.

2. Price Oracle Attack

This section provides a systematic overview of price oracle attacks and summarizes six known attacks in 2020.

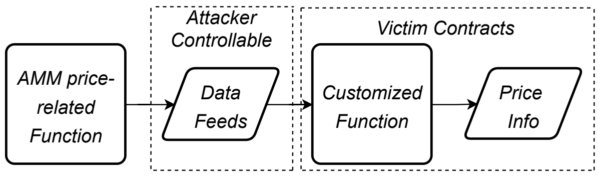

Figure 1. AMM-Based Price Oracle. Attackers can easily control the *data feeds* with flash loans and try to make profits from victim contracts.

Because price oracle attacks typically take advantage of flash loans to manipulate market prices on automated market makers, we also cover these concepts for ease of understanding. Readers interested in smart contracts and DeFi may refer to recent surveys [12], [13].

Automated Market Maker (AMM). Unlike the conventional market order books, AMMs execute trades on-chain automatically. There are mainly two types of traders in AMM, liquidity providers and liquidity takers. Liquidity providers deposit assets to the liquidity pool to earn exchange fees, while liquidity takers exchange assets by a predefined mathematical formula. One simple AMM model is the constant product market makers adopted by Uniswap [14], which keeps the product of the two asset reserves in a liquidity pool constant during trades.

Flash Loan. Flash loans are uncollateralized loans only valid within one blockchain transaction. Unlike traditional lending, flash loans allow traders to borrow any available amount of assets from the liquidity pool without upfront collateral, as long as the loan is repaid at the end of the transaction. If not, the transaction fails, and the execution is reverted as if the loan is never issued. As of April 2021, top flash loan providers include dYdX, Aave, and bZx [15]–[17]. Flash swaps, e.g., from Uniswap V2, similar to flash loans, allow users to withdraw up to the total reserves of the liquidity pools and repay them with a small fee by the end of the transaction [18].

Price Oracle. Price oracles are smart contracts that report the price of certain assets to DeFi protocols. Price oracles fetch price data from sources that are either off-chain or on-chain. This work focuses on on-chain oracles, which usually derive price information from the state of AMM liquidity pools. Several AMMs, including Uniswap, Curve, and Kyber Network [19], [20], have been used by numerous DeFi protocols to build on-chain price oracles. A price oracle can infer the market price of an asset with a customized function processing the price data feeds from *price-related functions* (Figure 1). For example, a liquidity pool of Uniswap V2 provides a function called *getReserves*, which returns the current reserves of the two tokens in the pool. A developer may write a customized function, such as $f(r_1, r_2) = r_1/r_2$, to calculate the ratio of the reserves, representing the price of one token to the other according to Uniswap's price model.

Price Oracle Attack: Overview This paper defines a *price oracle attack* as an attack that manipulates price oracles (i.e., controlling or biasing the reported price) to profit from the victim at a relatively low cost. Table 1 summarizes

TABLE 1. Summary of Price Oracle Attacks

Victim	Date	Flash Loan Provider	AMM	Price-Related Function	Target Asset	Loss
Warp Finance	2020.12.17	dYdX, Uniswap V2	Uniswap V2	*getReserves*	DAI-WETH-LP	$7.8M
Value DeFi	2020.11.14	Aave, Uniswap V2	Curve	*calc_token_amount*	USDC	$7.4M
Cheese Bank	2020.11.06	dYdX	Uniswap V2	*balanceOf*	CHEESE-WETH-LP	$3.3M
Plouto Finance	2020.10.29	Aave, Uniswap V2	Curve	*calc_withdraw_one_coin*	DAI	$700K
Harvest Finance	2020.10.26	Uniswap V2	Curve	*calc_withdraw_one_coin*	USDC, USDT	$24M
bZx	2020.02.18	bZx	Kyber	*getExpectedRate*	sUSD	$665.8K
			Uniswap V1	*getTokenToEthInputPrice*		
				getEthToTokenInputPrice		

six high-profile price oracle attacks in 2020 [6], [7], [21]. The victim DeFi protocols were Warp Finance, Value DeFi, Cheese Bank, Plouto Finance, Harvest Finance, and bZx. Notice that an attack may invoke multiple AMMs and price-related functions, but we only list those manipulated by the attacker. We will use these attacks in our evaluation.

The root causes of these attacks were the misuse of AMM price-related functions, such as implementing on-chain oracles that only reference a single source or calculating the target asset price using a formula vulnerable to manipulation. In these attacks, the attacker biased the AMM data feeds, and the bias was propagated by the vulnerable customized function, causing the victim to obtain inaccurate price information. Thus, the victim was deceived into overvaluing or undervaluing the target assets.

One mitigation to price oracle attacks is using time-weighted average price (TWAP) oracles, whose reported prices are more expensive to manipulate. The reason is that a typical on-chain TWAP oracle does not query the spot price (e.g., $getReserves$) from AMM functions but the cumulative price (e.g., $price0CumulativeLast$) with a last-recorded timestamp for calculating the time-weighted average price. To manipulate the cumulative price, an attacker must move the market price at the end of a block since it is updated at the beginning of each block according to the market price. However, because the attacker may be unable to arbitrage it back in the next block, the attacker risks the loss caused by price slippage [22].

Price Oracle Attack: Steps Price oracle attacks typically consist of four steps, all executed within the same transaction to prevent being interrupted by other users:

1) Preparing target assets: First, the attacker prepares the *target asset*, whose price he intends to inflate, from the victim or other DeFi protocols. He also needs to prepare additional funds for the next phase.

2) Inflating target assets price: The attacker manipulates the price oracles by imbalancing the reserves of the corresponding AMM liquidity pools, i.e., swapping a large number of tokens from one to another. Due to the oracle's vulnerable design, it is affected by the AMM state change and reports the manipulated price to the victim.

3) Profiting from the victim: This step is where the attacker gains profit during the whole attack. By exchanging the target assets for other assets through the victim's services (e.g., collateralized borrowing), the attacker gains profit since the victim overvalues the target assets.

4) Recovering target assets price: The attacker performs the reverse actions on the pools to restore the imbalanced AMM liquidity pools to their original state. As a result, the attacker avoids losses caused by the price slippage in the second step and gets back all his original assets, only paying for swap fees.

After performing the four steps described above, the attacker repaid the flash loans with optional borrowing fees. In practice, the attacker requires large capital to effectively shift the reserves in the liquidity pools, which is feasible by borrowing sufficient funds from one or multiple flash loans, as we have seen in past attack incidents. Appendix A provides a detailed example of these four steps.

3. Our Proposed Framework: ProMutator

ProMutator is a security analysis framework to assess whether a DeFi protocol is susceptible to price oracle vulnerabilities. At a high-level view, ProMutator simulates price oracle attacks locally and observes how a price oracle handles abnormal AMM price data feeds. We designed ProMutator with three objectives. First, it *proactively* discovers vulnerabilities in the target contracts instead of reactively reports the incidents after the attacks have occurred. Second, ProMutator is based on *transactions*; it requires the transactions sent to the target contracts but not their high-level source code (i.e., high-level code written in smart contract languages such as Solidity [23]). Third, it *flexibly* allows users to define mutation and detection rules to support various price-related functions and determine how the data feeds are mutated.

To use ProMutator, a user specifies a target contract to be analyzed and provides transactions to ProMutator as the input. The provided transactions can be either history transactions on the Ethereum Mainnet or testing transactions generated by the user. As shown in Figure 2, each transaction is processed through three phases: **Decoding**, **Mutation**, and **Analysis**. ProMutator comes with built-in mutation and detection rules for price oracle vulnerabilities. ProMutator can be extended to cover other types of vulnerabilities with custom mutation and detection rules. Finally, ProMutator outputs whether the target contract is vulnerable with a detailed report. We describe each phase in detail next.

3.1. Decoding

In this phase, the ABI of the target contract is used to decode the transactions to identify interesting input parameters. The decoded parameters are then passed to the next phase for mutation. As we will explain later, mutating input parameters can lead to the discovery of additional vulnerabilities. The phase is skipped for contracts without an available ABI.

3.2. Mutation

ProMutator identifies and mutates interesting values (which may affect the price oracle output) and simulates the execution to generate execution traces within our modified Go-Ethereum EVM [24]. The execution trace includes a summary of 1) functions called in the given transaction and 2) parameters provided [25]. A mutated trace refers to

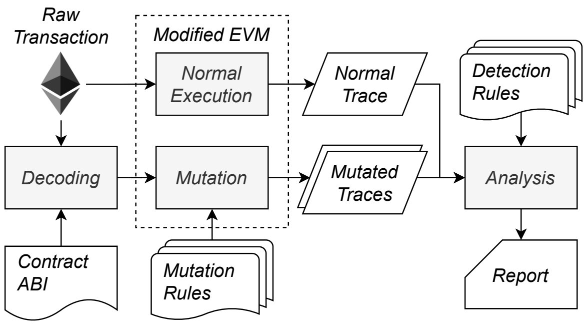

Figure 2. ProMutator's workflow

the execution trace of a mutated transaction. ProMutator currently supports the following two mutation rules, but more can be easily imported in the future if needed.

Return values of price-related functions. As mentioned in Section 2, the data feeds from AMMs can be easily manipulated with the assists of flash loans. Thus, we directly multiply or divide the return values of these functions by some mutation factor to simulate an attack without changing the AMM contracts' state. We choose mutation factors in the range of 1.2 to 10, which are not only theoretically possible but feasible in practice and supported by real-world incidents [26]. Take *getReserves* as an example, which returns two values, $a1$ and $a2$, with their product, $a1 \times a2$, being a constant. We multiply $a1$ by a chosen factor, say, 1.5, and divide $a2$ by 1.5 to simulate the imbalanced state of the liquidity pool. A complete list of mutated price-related functions with applied mutation operations is listed in Table 4.

Transaction input parameters. Many DeFi protocols require specifying the exact number of tokens to trade or operate in the function parameters, so we may be unable to observe the effect of the first mutation rule. Therefore, we mutate the input parameters of normal transactions to indicate the user's intention to operate with more or fewer tokens. All input parameters with type *int* or *uint* are multiplied by a mutation factor in the range of 0.1 to 10. For example, Warp Finance provides a function, *borrowSC(address,uint256)*, allowing users to borrow stable coins with the amount specified by the second parameter. We multiply this parameter by, for example, 0.5 or 2.5, to simulate the action of borrowing a smaller or larger number of coins and observe the changes.

In short, the first mutation rule simulates the price oracle attack, while the second rule decides the number of tokens to operate. The second mutation rule is primarily for the *Rule of Revert* detection rule (Section 3.3). The two mutation rules are independent of each other and can be applied simultaneously. If each interesting value is mutated five times, for a transaction with four price-related functions involved and two integer-type parameters, ProMutator generates one normal trace and $(4\times5+1)\times(2\times5+1)-1 = 230$ mutated traces. The more mutated traces, the more accurately ProMutator detects a vulnerability, but with lower efficiency as a trade-off since the number of mutated traces is proportional to the total runtime.

3.3. Analysis

In this phase, the normal execution trace and the mutated ones are compared with each others to determine whether the target contract is vulnerable or not, based on detection rules. For price oracle vulnerabilities, we specify two detection rules.

TABLE 2. EVALUATION ON PAST ATTACKS
✓: VULNERABILITY IS DETECTABLE, ✗: UNDETECTABLE

Victim	RT	RR	Vulnerable Functions
Warp Finance	✓	✓	*borrowSC, withdrawCollateral*
Value DeFi	✓	✓	*deposit, withdraw*
Cheese Bank	✗	✗	N/A
Plouto Finance	✓	✗	*deposit, withdraw*
Harvest Finance	✓	✗	*deposit, withdraw*
bZx	✗	✓	*borrowTokenFromDeposit*

Rule of Transfer (RT) The *Rule of Transfer* detects whether the transfer value of ETH or any ERC20 token changes due to mutation, which indicates that the price data feeds affect the number of tokens transferred, including those minted or burned. Since one of the most crucial concerns of smart contract security is to prevent the attacker from controlling or influencing the token transfers between contracts, the *Rule of Transfer* detects the vulnerability with high confidence and accuracy.

Rule of Revert (RR) The *Rule of Revert* only focuses on transactions whose input parameters are mutated. If a transaction fails after mutating the input parameters but becomes successful after also mutating the data feeds from AMMs, then the target contract is considered vulnerable by the Rule of Revert. The Rule of Revert is designed to detect vulnerable functions that the Rule of Transfer cannot, such as some in Warp Finance, Value DeFi, and bZx. A more detailed explanation with an example is provided in Appendix A.

ProMutator supports contracts with and without an ABI. With ABI, ProMutator can additionally recover and mutate integer-type parameters and thus apply the Rule of Revert. When the transaction input parameters are not mutated, ProMutator applies the Rule of Transfer. Otherwise, ProMutator applies the Rule of Revert. The target contract is flagged vulnerable if either detection rule is met, and ProMutator outputs a report specifying the mutation details. An example attack report is presented in Appendix C.

4. Evaluation

ProMutator examines whether a DeFi protocol is susceptible to price oracle vulnerabilities. To evaluate ProMutator's effectiveness, we applied ProMutator to analyze 1) the transactions predating known attack incidents and 2) transactions from block 11,090,000 to 11,490,000 (Oct.–Dec. 2020) on the Ethereum Mainnet. We set up an Ethereum node on an Ubuntu device with AMD Ryzen 9 3950X, 2TB NVMe SSD, and 256GB RAM. In the second part of the evaluation, we collected 5,964 contracts and randomly picked four transactions for each contract. If a contract called any of the AMM price-related functions listed in Table 3, whether directly or indirectly, this contract was identified as using AMM price feeds and evaluated in our experiment. The runtime of evaluating all transactions was within 24 hours (3.62 seconds per transaction), which was an acceptable amount of time.

4.1. Past Attacks

Table 2 shows the evaluation results on the six attack incidents (see Table 1 for a summary of these incidents). ProMutator successfully rediscovered that five out of six victims (except for Cheese Bank) were vulnerable to price oracle attacks, with their vulnerable functions shown in the column *Vulnerable Functions*. We further manually confirmed that the produced attack reports are valid.

TABLE 3. Evaluation on Undisclosed Vulnerabilities

AMM Type	Total	Sampled	Unique	TP	FP	Precision
Uniswap	176	20	17	5	12	29.41%
Curve	39	39	29	21	8	72.41%
Kyber	4	4	4	1	3	25.00%

Among the past attacks, the only undetectable vulnerability was that of Cheese Bank, which updated token prices in a two-stage manner. The fetched prices were stored in state variables first, and read from the variables each time when needed. Unless the function *refresh* is explicitly called, the token prices will not be updated from the oracle. While developers and benign users did not call *refresh* before calling *borrow* of Cheese Bank, a successful attack should. ProMutator could not find the vulnerability of Cheese Bank because it could not recover such combination of actions from benign transactions.

4.2. Undisclosed Vulnerabilities

Apart from those known attack incidents in the past, we evaluated ProMutator on existing DeFi protocols and identified a total of 219 potentially vulnerable price oracles, with 27 of them confirmed vulnerable by manual examination. (Table 3).

We manually evaluated all potential vulnerabilities built on Curve and Kyber and 10% of those built on Uniswap to determine whether they are true or false positives. A complete evaluation list with vulnerability details will be published on our official vulnerability disclosure platform [27] in accordance with our responsible disclosure plan.

Most of the ProMutator's false positives were cases in which the examined protocols directly operate (e.g., swapping) on the referenced AMM pools and thus affected their state. The attacker may suffer losses if the protocol's operation reduces the price spread caused by the attacker previously, and the losses may be higher than the swap fees. Thus, we consider the price oracle attack unsuccessful according to our definition. We leave it as future work to support the detection of AMM state-changing operations such that ProMutator can rule out such false positives.

ProMutator performs the best in vulnerabilities built on Curve, which were mostly price oracles for vaults. The compromise of a vault's price oracle could lead to significant loss of users' funds, as shown in the Harvest, Plouto, and Value DeFi attacks. Moreover, ProMutator correctly reported that TWAP oracles are not vulnerable to price manipulations (i.e., true negatives) because we exclude functions that are expensive to manipulate (e.g., price accumulators, commonly used by TWAP oracles) from our list of to-be-mutated AMM price-related functions.

5. Related Work

Security issues in smart contracts have attracted significant attention since the DAO attack [28] and the Parity multi-sig wallet attack [29] occurred in the early days of Ethereum [30]. To find smart contract vulnerabilities, researchers have applied automated software testing techniques such as symbolic execution [12], [31]–[34], fuzzing [10], [35], static analysis [36], [37], and formal verification [38]. These tools can efficiently spot critical vulnerabilities within individual contracts, and many of them have become built-in functionalities of modern smart contract development frameworks [39]. However, applying existing analysis tools on a combination of contracts would scale poorly due to numerous possible contract combinations.

Several approaches [4], [5], [40], [41] use existing transactions to narrow down possible combinations for improved scalability. Wu et al. [9] converted existing transactions to a high-level semantic to detect past price oracle attacks. The idea of using existing transactions has inspired our work. Our tool can discover price oracle vulnerabilities before attackers exploit them, whereas previous approaches are limited to conduct post-incident analysis.

A line of research focuses on analyzing or modeling DeFi composability [3], [6], [7]. Zhou et al. [3] applied an SMT solver to find profitable actions interacting with several contracts. Wang et al. [8] performed symbolic reasoning on oracles and data flow analysis to identify oracle-dependent state updates. Tolmach et al. [42] applied a process-algebraic approach to model DeFi protocols in a compositional manner for efficient property verification. Our work differs from previous works in that ProMutator finds vulnerabilities by locally simulating composable actions instead of modeling the DeFi components.

At a high-level view, ProMutator falls into the category of grey box testing and differential testing. ProMutator applies grey box testing since it knows valid user inputs and mutates them to fuzz price oracles. Besides, given the same transaction input, ProMutator observes the difference between a normal and mutated trace.

6. Discussion, Conclusion, and Future Work

Aiming at a more generic and lightweight approach, we chose to directly mutate the AMM data feeds instead of simulating the attack using an actual flash loan. Although a full simulation could increase ProMutator's accuracy, it would require additional engineering efforts and protocol-specific implementation because ProMutator would have to prepare corresponding tokens to imbalance a particular AMM pool. For example, imbalancing the Curve Y pool requires obtaining yTokens from the Yearn protocol beforehand [43]. Thus, we assume AMM pools have been imbalanced and focus on mutating the return values of the AMM price-related functions.

ProMutator outperforms general smart contract fuzzers in finding price oracle vulnerabilities since it does not require the examined contracts' high-level source code and supports various price oracles without manual intervention. As far as we know, well-known and open-sourced general fuzzers require knowing high-level source code, which may be unavailable or depend on additional libraries to be compiled before fuzzers could run. Moreover, it is difficult for general fuzzers to determine whether a price oracle is vulnerable without knowing its behavior under normal circumstances. On the other hand, ProMutator "learns" the expected behavior of price oracles by examining the given transactions, which is comparably more automated and convenient to use.

This paper demystified price oracle attacks and proposed ProMutator to detect whether a DeFi protocol is susceptible to price oracle attacks by examining and mutating transactions. Our future work is to extend ProMutator to cover more price oracle vulnerabilities, e.g., those in the Cheese Bank Attack. This may be achieved by propagating the mutation effect from one transaction to another, identifying the affected state variables during a mutation, and providing additional mutation rules for state variables.

References

[1] "Defi pulse - the decentralized finance leaderboard," https://defipulse.com, accessed: 2021-05-22.

[2] L. Gudgeon, D. Perez, D. Harz, A. Gervais, and B. Livshits, "The decentralized financial crisis: Attacking defi," *CoRR*, vol. abs/2002.08099, 2020. [Online]. Available: https://arxiv.org/abs/2002.08099

[3] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," *CoRR*, vol. abs/2103.02228, 2021. [Online]. Available: https://arxiv.org/abs/2103.02228

[4] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "TXSPECTOR: Uncovering attacks in ethereum from transactions," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2775–2792.

[5] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "SODA: A generic online detection framework for smart contracts," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[6] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the defi ecosystem with flash loans for fun and profit," 2021.

[7] Y. Cao, C. Zou, and X. Cheng, "Flashot: A snapshot of flash loan attack on defi ecosystem," 2021.

[8] B. Wang, H. Liu, C. Liu, Z. Yang, Q. Ren, H. Zheng, and H. Lei, "Blockeye: Hunting for defi attacks on blockchain," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 17–20.

[9] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He, and K. Ren, "Defiranger: Detecting price manipulation attacks on defi applications," 2021.

[10] G. Grieco, W. 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*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560. [Online]. Available: https://doi.org/10.1145/3395363.3404366

[11] "Mythril," https://github.com/ConsenSys/mythril, accessed: 2021-05-13.

[12] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269.

[13] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (defi)," *arXiv preprint arXiv:2101.08778*, 2021.

[14] "Uniswap," https://uniswap.org/, accessed: 2021-05-13.

[15] "dydx," https://dydx.exchange/, accessed: 2021-05-13.

[16] "Aave," https://aave.com/, accessed: 2021-05-13.

[17] "bzx," https://bzx.network/, accessed: 2021-05-13.

[18] "Flash swaps," https://uniswap.org/docs/v2/core-concepts/flash-swaps/, accessed: 2021-05-13.

[19] "Curve," https://curve.fi/, accessed: 2021-05-13.

[20] "Kyber network," https://kyber.network/, accessed: 2021-05-13.

[21] "Plouto was attacked by flashloan," https://ploutoprotocol.medium.com/plouto-was-attacked-by-flashloan-c309161c6281, accessed: 2021-05-13.

[22] "Oracles," https://uniswap.org/docs/v2/core-concepts/oracles/, accessed: 2021-05-13.

[23] "Solidity," https://docs.soliditylang.org/en/v0.8.6/, accessed: 2021-06-29.

[24] "Go ethereum," https://geth.ethereum.org/, accessed: 2021-05-13.

[25] "Evm tracing," https://geth.ethereum.org/docs/dapp/tracing, accessed: 2021-06-29.

[26] "Analysis for cheese bank incident's attack transaction," https://ethtx.info/mainnet/0x600a869aa3a259158310a233b815ff67ca41eab8961a49918c2031\297a02f1cc, accessed: 2021-06-29.

[27] "Promutator," https://github.com/csienslab/ProMutator, accessed: 2021-07-07.

[28] D. Siegel, "Understanding the dao hack for journalists," June 2016. [Online]. Available: https://pullnews.medium.com/understanding-the-dao-hack-for-journalists-2312dd43e993#.kw0ufw25q

[29] B. S. Palladino, S. Palladino, O. Security, and OpenZeppelin, "The parity wallet hack explained," May 2020. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[30] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Berlin, Heidelberg: Springer-Verlag, 2017, p. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[31] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[32] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 653–663. [Online]. Available: https://doi.org/10.1145/3274694.3274743

[33] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[34] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[35] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[36] 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 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[37] 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*, 2018, pp. 9–16.

[38] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[39] "Solidity visual developer," https://github.com/ConsenSys/vscode-solidity-auditor, accessed: 2021-05-13.

[40] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[41] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of horus: Spotting and analyzing attacks on ethereum smart contracts," 2021.

[42] P. Tolmach, Y. Li, S.-W. Lin, and Y. Liu, "Formal analysis of composable defi protocols," *arXiv preprint arXiv:2103.00540*, 2021.

[43] "Yearn finance," https://yearn.finance/, accessed: 2021-06-29.

[44] "Warpfinance incident: Root cause analysis," https://blog.peckshield.com/2020/12/18/warpfinance/, accessed: 2021-05-13.

TABLE 4. AMM PRICE-RELATED FUNCTIONS WITH MUTATION RULES

Price Data Source	Component	Price-Related Function	Return Value Type	Mutation Operation

UniswapV1 [14]	Pair	*getEthToTokenInputPrice(uint256)*
		getEthToTokenOutputPrice(uint256)
		getTokenToEthInputPrice(uint256)
		getTokenToEthOutputPrice(uint256)
			uint256	Multiply

UniswapV2 [14]	Pair	*getReserves()*	*uint256,uint256*	Multiply, Divide

Curve Finance [19]	StableSwap	*calc_token_amount(uint256[2],bool)*
		calc_token_amount(uint256[3],bool)
		calc_token_amount(uint256[4],bool)
			uint256	Multiply

		calc_withdraw_one_coin(uint256,int128)	*uint256*	Multiply
	DepositZap	*calc_token_amount(address,uint256[4],bool)*	*uint256*	Multiply
		calc_withdraw_one_coin(address,uint256,int128)	*uint256*	Multiply
Kyber Network [20]	Network Proxy	*getExpectedRate(address,address,uint256,bool)*	*uint256,uint256*	Both Multiply

Appendix A. An Example of Price Oracle Attack

We take the Warp Finance incident as an example to further illustrate the details of a price oracle attack. On Dec. 17, 2020, Warp Finance was attacked [44] by the attacker artificially inflating the price of the DAI-WETH Uniswap V2 LP token, which was then provided to Warp Finance as the collateral to borrow DAI.

This attack consisted of four steps. First, the attacker borrowed WETH and DAI from dYdX flash loan services and three Uniswap V2 flash swaps, adding his DAI and a portion of WETH to the Uniswap V2 DAI-WETH pool to mint LP tokens. He then provided all his LP tokens to Warp as collateral, whose price was 58.8 USD before manipulation. Next, he swapped the rest of his WETH to DAI to skew the reserves in the liquidity pool, causing the LP token price to rise to 135.5 USD due to the vulnerable pricing formula Warp used. Lastly, he borrowed DAI and USDC from Warp at a high price of LP tokens, repaid the flash loan and flash swaps, left the loan underwater, and walked away with 7.8 million dollars.

The vulnerability of Warp Finance's price oracle was its pricing formula

$$p_{LP} = \frac{p_0 r_0 + p_1 r_1}{total_{LP}}$$

that calculated the price of Uniswap V2 DAI-WETH-LP tokens, where p_0, r_0, p_1, and r_1 were the price and reserves of DAI and WETH, respectively. Though the prices were calculated using a Uniswap's TWAP oracle, the reserves were fetched directly from the liquidity pool and thus were vulnerable to manipulation.

Appendix B. An Example Detected by Rule of Revert

The function $borrowSC(token, amount)$ calculates the value of the user's collateral with the above vulnerable pricing formula to determine whether the borrowing is allowed. In the Warp attack, the attacker exploited this function to borrow more stable coins than he could by manipulating the price of collateral he provided. Although this function is vulnerable, it cannot be detected by the Rule of Transfer due to its parameter design. Consider a normal transaction calling *borrowSC(DAI, 100)*. This transaction always transfers 100 DAI to the borrower, though the collateral price has been inflated.

The purpose of mutating the input parameters and applying the Rule of Revert is to reduce false negatives as in this situation. For example, by mutating the second parameter to a large value (e.g., 1,000), the transaction may fail since the user does not provide enough collateral for borrowing that much DAI. However, if the transaction becomes successful by additionally mutating the AMM data feeds, the target function is likely to be vulnerable since the transaction's success depends on the AMM's state. Therefore, the Rule of Revert identifies this situation and correctly marks this function as vulnerable.

Appendix C. An Example of Attack Report

The attack report of Value DeFi protocol is listed as follows. Value DeFi is identified vulnerable by a transaction calling the function *withdraw*, according to the Rule of Transfer. The report points out that the price data feed from the two AMM pools, *Curve 3pool* and *Curve BUSD Deposit*, could affect the transfer amount of *3CRV*.

```
Analyzing transaction: 0xa07d381e
Entry function: ValueMultiVaultBank.withdraw (0x23deedcf)

Found 2 price-related functions:
  - Curve_BUSD_Deposit.calc_withdraw_one_coin
  - Curve_3pool.calc_token_amount
Collected 2 Transfer events:
  - VALUE.Transfer
  - 3CRV.Transfer

Found 2 successful mutations:
Function: Curve_BUSD_Deposit.calc_withdraw_one_coin
Multiply factor: 2.0
Detected: Rule of Transfer
  - 3CRV transfer amount changed
    - Before: 1000937998334335426
    - After: 1231650125499646858
Attack:
  - Swap in USDC to Curve_BUSD_Deposit
  - Call ValueMultiVaultBank.withdraw

Function: Curve_3pool.calc_token_amount
Multiply factor: 2.0
Detected: Rule of Transfer
  - 3CRV transfer amount changed
    - Before: 1000937998334335426
    - After: 1264652768033323364
Attack:
  - Swap out USDC from Curve_3pool
  - Call ValueMultiVaultBank.withdraw
```