Automated Attack Synthesis for Constant Product Market Makers

SUJIN HAN, KAIST, Republic of Korea
JINSEO KIM, KAIST, Republic of Korea
SUNG-JU LEE, KAIST, Republic of Korea
INSU YUN, KAIST, Republic of Korea

Decentralized Finance (DeFi) enables many novel applications that were impossible in traditional finances. However, it also introduces new types of vulnerabilities. An example of such vulnerabilities is a composability bug between token contracts and Decentralized Exchange (DEX) that follows the Constant Product Market Maker (CPMM) model. This type of bug, which we refer to as CPMM composability bug, originates from issues in token contracts that make them incompatible with CPMMs, thereby endangering other tokens within the CPMM ecosystem. Since 2022, 23 exploits of such kind have resulted in a total loss of 2.2M USD. BlockSec, a smart contract auditing company, reported that 138 exploits of such kind occurred just in February 2023.

In this paper, we propose *CPMMX*, a tool that automatically detects CPMM composability bugs across entire blockchains. To achieve such scalability, we first formalized CPMM composability bugs and found that these bugs can be induced by breaking two safety invariants. Based on this finding, we designed *CPMMX* equipped with a two-step approach, called shallow-then-deep search. In more detail, it first uses shallow search to find transactions that break the invariants. Then, it uses deep search to refine these transactions, making them profitable for the attacker. We evaluated *CPMMX* against five baselines on two public datasets and one synthetic dataset. In our evaluation, *CPMMX* detected 2.5x to 1.5x more vulnerabilities compared to baseline methods. It also analyzed contracts significantly faster, achieving higher F1 scores than the baselines. Additionally, we applied *CPMMX* to all contracts on the latest blocks of the Ethereum and Binance networks and discovered 26 new exploits that can result in 15.7K USD profit in total.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering**;

Additional Key Words and Phrases: Smart Contract, Security, Composability, Exploit Generation

ACM Reference Format:
Sujin Han, Jinseo Kim, Sung-Ju Lee, and Insu Yun. 2025. Automated Attack Synthesis for Constant Product Market Makers. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA003 (July 2025), 22 pages. https://doi.org/10.1145/3728872

1 Introduction

Decentralized Finance (DeFi) provides new financial services using blockchain and smart contracts. These services use tokens, which are digital assets beyond native currencies on the blockchain. A key financial service smart contracts offer is Decentralized Exchanges (DEX). Unlike Centralized Exchanges (CEX), DEXes enable users to swap one asset for another without a central authority. Through DEXes, blockchain users can freely convert their assets, which provides fluidity in the blockchain economy. To enable swapping without an intermediary, most DEXes adopt the Constant Product Market Maker (CPMM) model to automatically determine appropriate exchange rates.

Authors' Contact Information: Sujin Han, KAIST, Daejeon, Republic of Korea, sujinhan@kaist.ac.kr; Jinseo Kim, KAIST, Daejeon, Republic of Korea, jinseo@kaist.ac.kr; Sung-Ju Lee, KAIST, Daejeon, Republic of Korea, profsj@kaist.ac.kr; Insu Yun, KAIST, Daejeon, Republic of Korea, insuyun@kaist.ac.kr.

ACM 2994-970X/2025/7-ARTISSTA003
https://doi.org/10.1145/3728872

This DeFi ecosystem is often threatened by new types of vulnerabilities, one of which is CPMM composability bugs. CPMM composability bugs is a composability bug [5] arising from the interaction between a CPMM DEX and a token contract. This type of vulnerability allows an attacker to steal assets from a flawless DEX by exploiting a bug in a token contract. Recently, this type of vulnerability has been frequently exploited. For instance, on January 20, 2023, an attacker leveraged BRA token's flawed tax mechanism to steal around 225K USD worth of digital assets from a DEX [26]. Moreover, BlockSec, which is a renowned security auditing company, reported 138 attacks of such kind just in February 2023 [6].

Several tools [17, 29, 33] have been developed for multi-contract bug detection, yet detecting CPMM composability bugs remains challenging. First, existing tools suffer from a large search space because they target multiple types of vulnerabilities. Second, these tools are unsuitable for generating profitable transactions (i.e., end-to-end exploits) because most rely on coverage-guided fuzzing. Coverage-guided fuzzing focuses on new coverage rather than making profits, which often requires multiple repetitions of internal calls. Lastly, these tools mostly rely on timeouts without offering early termination. This property makes them unsuitable for scanning vulnerabilities across entire blockchains, where benign contracts are predominant.

To address these challenges, we propose *CPMMX*, a tool that automatically detects CPMM composability bugs and synthesizes profitable transactions for the detected vulnerabilities. We first formalize CPMM composability bugs and identify that these bugs are caused by two broken invariants between a token contract and a CPMM DEX. Based on this, *CPMMX* employs a two-step approach, called *shallow-then-deep search*, to detect CPMM composability bugs. In the shallow search, *CPMMX* attempts to discover transactions that break the invariants. If such transactions are found, *CPMMX* refines them in the deep search to make them profitable.

We compared *CPMMX* against five baseline tools, Echidna [17], Ityfuzz [29], DeFiTainter [21], Slither [13], and Mythril [27], on two public and one synthetic datasets. In our evaluation, *CPMMX* outperformed existing tools, detecting 2.5× and 1.5× more vulnerabilities on the two public exploit datasets. On the synthetic dataset, it achieved an F1 score of 0.97, compared to 0.66 for the next best tool, ItyFuzz [29]. Notably, *CPMMX* completed this analysis in 10 hours, which is 6.9× faster than ItyFuzz. Furthermore, to demonstrate the effectiveness of *CPMMX* in the real world, we ran it on Ethereum and Binance. It discovered 26 profitable transactions, which can yield 15.7K USD profit.

To summarize, we make the following contributions:

- We formalize CPMM composability bugs and identify two safety invariants that, when broken, allow an attacker to steal funds from DEXes.
- We design and implement *CPMMX* that automatically detects CPMM composability bugs across entire blockchains. It employs a novel approach, *shallow-then-deep* search, to efficiently identify CPMM composability bugs without false positives.
- We evaluate *CPMMX* on several datasets and compare it with five baseline tools. Moreover, we demonstrate its effectiveness in the real world by running it on Ethereum and Binance; it identified **26 undiscovered vulnerabilities** that can yield **total 15.7K USD profit.**

2 Background

ERC20 tokens. Tokens are digital assets on the blockchain. Among these tokens, the most commonly used are fungible tokens referred to as ERC20 tokens, defined through the Ethereum Request for Comments 20 (ERC20).[1] Native currencies (e.g., ETH in Ethereum or BNB in Binance) can also

[1]Although each blockchain may refer to them differently according to their protocol (e.g., BEP20 or TRC20), we collectively call them as ERC20 Tokens in this paper.

utilize ERC20 services through wrapper implementations, such as Wrapped Ethereum (WETH) or Wrapped Binance Coin (WBNB).

The ERC20 standard requires a token smart contract to implement a set of Application Binary Interface (ABI) consisting of 9 functions and 2 events. These functions are necessary for basic operations of tokens, such as `transfer(address,value)` and `balanceOf(address)`. Such a uniform interface allows developers to build financial services, such as DEXes, for countless tokens without having to write custom code for each token. This design also increases the flexibility of ERC20 token implementation, as developers can freely implement each ABI function. However, it also increases the risk of potentially violating critical safety invariants within a service.

Constant Product Market Maker model. The Constant Product Market Maker (CPMM) model is adopted by DEXes to automatically swap one ERC20 token for another ERC20 token at an appropriate exchange rate. The CPMM model states that, given a DEX holding x amount of X tokens and y amount of Y tokens, the product of x and y should remain the same (i.e., $x \times y = k$). When a user requests to swap Δx amount of X tokens for Y tokens, the amount of Y tokens the DEX returns, Δy, is calculated with the equation $(x + \Delta x) \times (y - \Delta y) = k$. Thus, any swap operation in a CPMM DEX can be represented as a movement along the curve $x \times y = k$.

The majority of DEXes today charge a small percent fee for each exchange to provide profit for the liquidity providers who deposited the initial X and Y tokens. For example, the Uniswap protocol [3], which is the most widely used DEX, charges a 0.3% fee for each exchange. As a result, most DEXes can be said to have adopted a modified version of the CPMM model, where the product of two assets slightly increases after each exchange (i.e., $x \times y \geq k$).

3 Motivation

```
1 uint public rewardRate = 5;
2 uint public percent = 10000;
3 uint public minAmount = 10000 * 1e18;
4 function giveReward(address receiver, uint amount)
      private {
5   if (amount > minAmount) {
6     rewardAmount = amount * rewardRate / percent;
7     balances[receiver] += rewardAmount;
8   }
9 }
10 function transfer(address sender, address receiver,
      uint amount) public {
11   if (sender == DEX_ADDR) {
12     giveReward(receiver, amount);
13   } else if (receiver == DEX_ADDR) {
14     giveReward(sender, amount);
15   }
16   balances[sender] -= amount;
17   balances[receiver] += amount;
18 }
```

Fig. 1. Vulnerable code snippet in ANCH token.

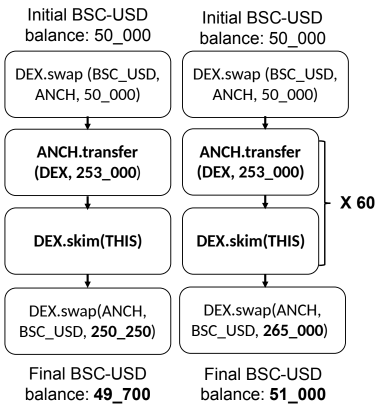
Initial BSC-USD balance: 50_000
Initial BSC-USD balance: 50_000
DEX.swap (BSC_USD, ANCH, 50_000)
DEX.swap (BSC_USD, ANCH, 50_000)
ANCH.transfer (DEX, 253_000)
ANCH.transfer (DEX, 253_000)
X 60
DEX.skim(THIS)
DEX.skim(THIS)
DEX.swap(ANCH, BSC_USD, 250_250)
DEX.swap(ANCH, BSC_USD, 265_000)
Final BSC-USD balance: 49_700
Final BSC-USD balance: 51_000

Fig. 2. ANCH token exploit without repetition (left) and with repetition (right).

3.1 Motivating Example

We present a motivating example that illustrates the challenges of detecting bugs in smart contracts and the need for a new approach to identify them effectively. On August 9, 2022, an attacker exploited the ANCH token contract to extract approximately 19.9K USD worth of stablecoins from the ANCH-BSC-USD DEX [4]. This happened because the attacker could increase their ANCH token balance without making any payment. Figure 1 shows a simplified version of the vulnerable code. In lines 11 and 13, the token contract checks whether the sender or receiver of the transfer is the DEX contract; if so, it rewards 0.05% of the transfer amount to the receiver or sender, as shown in lines 6-7. Exploiting this behavior, the attacker accumulated many ANCH tokens to nearly drain the ANCH-BSC-USD DEX.

Unfortunately, the ANCH exploit cannot be easily detected with existing tools. The key obstacle is that this bug needs to be triggered multiple times to generate profit. Existing fuzzers may produce test cases similar to the one shown on the left of Figure 2, which triggers the bug but is not profitable. The attacker can gain rewards from `ANCH.transfer(DEX, 253_000)` and also `DEX.skim(THIS)`. Notably, `skim` is a function that makes the DEX send extraneous tokens to the address given as the argument, internally calling `ANCH.transfer(THIS, 253_000)`. Since the ANCH token rewards only 0.05% of the transfer amount, receiving the reward twice is not enough to offset the swap fees, which are typically 0.3%. Moreover, it is not desirable to flag any reward mechanism like this as vulnerable; many benign tokens exhibit similar behaviors to incentivize users. Thus, building a profit-generating test case similar to the one on the right of Figure 2 is essential. However, existing tools struggle to generate such test cases because they require multiple repetitions of the reward-reaping call to generate profit. Existing tools like Echidna [17] or ItyFuzz [29], inspired by the success of coverage-guided fuzzing in traditional software (e.g., AFL [16]), use guidance strategies that aim to maximize code coverage. Unfortunately, such guidance strategies are ineffective at detecting this type of vulnerability because repetition does not improve code coverage but makes incremental changes in contract states. On a more practical note, these tools require specific contracts for testing. Consequently, lesser-known contracts like ANCH could remain untested.

3.2 Prevalence of CPMM Composability Bugs

Table 1. CPMM composability bugs found in the DeFiHackLabs dataset.

Vulnerable Token	Invariant Broken	Date of Exploit	Reported Loss	Reported Loss in USD
Wdoge	1	2022/04/24	78.6 BNB	30.2 K
LPC	2	2022/07/25	45.1 K BSC-USD	45.1 K
ANCH	2	2022/08/09	19.9 K BSC-USD	19.9 K
XST	2	2022/08/10	27.4 ETH	46.2 K
Shadowfi	1	2022/09/02	1.08 K BNB	300 K
PLTD	1	2022/10/18	24.5 K BSC-USD	24.5 K
HEALTH	1	2022/10/20	16.6 BNB	4.54 K
AES	1	2022/12/07	61.6 K BSC-USD	61.6 K
BGLD	1	2022/12/12	8.80 BNB	2.40 K
BRA	2	2023/01/10	228K BSC-USD	228 K
Upswing	1	2023/01/18	22.6 ETH	35.6 K
ThoreumFi	2	2023/01/19	2.26 K BNB	659 K
SHEEP	1	2023/02/10	9.54 BNB	2.93 K
Starlink	1	2023/02/17	38.4 BNB	11.8 K
BIGFI	1	2023/03/22	30.3 K BSC-USD	30.3 K
GPT	1	2023/05/25	155K BSC-USD	155 K
Bamboo	1	2023/07/04	235 BNB	57.6 K
ApeDAO	1	2023/07/18	19.2 K BSC-USD	19.2 K
HCT	1	2023/09/07	30.5 BNB	6.58 K
BFC	1	2023/09/09	42.3 K BSC-USD	42.3 K
pSeudoEth	2	2023/10/08	1.44 ETH	2.34 K
TGBS	1	2024/03/06	377 BNB	154 K
GHT	1	2024/03/07	15.4 ETH	58.6 K

We refer to bugs that affect CPMM DEXes due to vulnerabilities in token contracts, like the ANCH exploit, as *CPMM composability bugs*. Recently, attackers have frequently exploited CPMM composability bugs to extract considerable amounts of tokens from DEXes. For example, BlockSec [7], a renowned smart contract auditing firm, reported that CPMM composability bugs caused 138 exploits in February 2023. We could also find CPMM composability bugs in DeFiHackLabs [12], a public exploit replication dataset. The first and second authors manually inspected all exploits in the DeFiHackLabs dataset (as of March 2024) to create a subset where CPMM composability bugs cause the exploits. We found a total of 23 exploits and reported the details in Table 1. We categorized them based on the specific invariant broken, which is explained in §5. The cumulative financial loss of the 23 exploits is 2.2M USD.

CPMM composability bugs pose a significant threat to financial assets stored on the blockchain because CPMMs are widely used to facilitate token exchanges and manage substantial financial assets. As of June 2023, DEXes following the CPMM model were reported to take up around 77% of the market share, representing around 35.4 billion USD [22]. Therefore, identifying CPMM composability bugs is critical to ensuring the security of DeFi.

3.2.1 Trend Analysis. To illustrate the trend of CPMM composability bugs, we report their monthly proportion and financial losses using the DeFiHackLabs dataset [12]. As shown in Figure 3, these

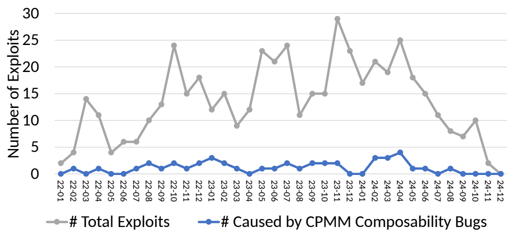
Fig. 3. Total exploits and CPMM composability bug exploits per month in the DeFiHackLabs dataset.

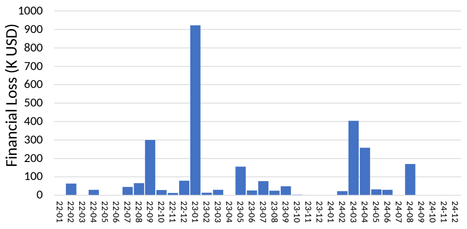
Fig. 4. Total financial loss per month due to CPMM composability bugs in the DeFiHackLabs dataset.

bugs have been consistently reported from February 2022 to August 2024. We expect more CPMM composability bugs to be reported for the last quarter of 2024, as incidents typically take a few months to be incorporated into the dataset. As shown in Figure 4, CPMM composability bugs still incur significant financial losses. For example, the ARK exploit (March 2024) caused a loss of 192K USD, while the IvestDao exploit (August 2024) resulted in 170K USD in losses.

4 Goals and Approaches

This section discusses the goals and approaches for building a tool, *CPMMX*, to detect CPMM composability bugs across all blockchains.

4.1 Detecting CPMM Composability Bugs Across Entire Blockchains

As shown in §3.2, CPMM composability bugs have been exploited repeatedly, posing significant threats to financial assets on the blockchain. However, existing tools are insufficient to detect CPMM composability bugs at scale. First, as explained in §3.1, the search strategies used by current tools are not well-suited to detect these bugs. Second, existing tools have limited scalability. Running them on the entire blockchain is computationally infeasible because they require significant computational resources to thoroughly test each contract. Therefore, there is a need for a tool capable of detecting CPMM composability bugs across all smart contracts on the blockchain.

Our approach: formalizing CPMM composability bugs and building a tool to detect them. To detect real-world vulnerabilities on a large scale, we decided to focus on CPMM composability bugs and built a tool, named *CPMMX*, to automatically detect it. *CPMMX* is designed to operate on the entire blockchain by minimizing computational costs. This is achieved by focusing the search space on areas likely to contain CPMM composability bugs and implementing early termination for benign contracts. This was possible because CPMM composability bugs exhibit properties that make them particularly suitable for automated detection. In §5, we formalize this vulnerability and describe how we can detect it by checking invariant violations.

4.2 Efficiently Detecting CPMM Composability Bugs

Even after limiting our scope to CPMM composability bugs, it is still challenging to analyze numerous smart contracts. As we can see from the motivating example in §3.1, we need to build a complicated transaction with a long sequence of internal calls to exploit CPMM composability bugs. Therefore, finding this bug by searching naïvely requires an impractical amount of computation.

Our approach: shallow-then-deep search. To address this problem, we propose a technique called *shallow-then-deep search. CPMMX* searches for CPMM composability bugs in two phases. The first phase, called shallow search, quickly identifies candidate contracts that may be vulnerable. However, real-world smart contracts are diverse and complex, making distinguishing between signs of vulnerabilities and intended behaviors difficult. To address this, *CPMMX* performs a second phase, called deep search, which explores the remaining contracts in more detail. In this phase, *CPMMX* generates a profitable transaction, which can be proof of the vulnerability. This approach allows us to analyze all smart contracts efficiently and detect CPMM composability bugs.

4.3 Minimizing False Positives

Eliminating false positives is crucial for effectively detecting vulnerabilities across all smart contracts. Even with a low false positive rate, analyzing numerous smart contracts could still result in hundreds or thousands of false positive cases, placing a significant burden on analysts.

Our approach: calculating profit using stablecoins and native currencies. To address this, *CPMMX* automatically generates end-to-end profitable transactions. Unlike existing methods that calculate profit by approximating the value of coins (e.g., Midas [33]), our approach adjusts transactions to ensure that all outcomes converge into tokens with relatively stable values (i.e., stablecoins or native currencies). *CPMMX* then computes the profit by evaluating the increase in these coins. This method is more reliable than previous approaches, which led to false positives due to noise in the value-based profit calculations. In contrast, *CPMMX* allows users to analyze based on clear financial gains, minimizing ambiguity and false positives.

5 Formalizing CPMM Composability Bugs

In this section, we define and explain CPMM composability bugs. First, we provide formal definitions in §5.1. Then, we explain how they can be utilized for profit with example exploits in §5.2 and §5.3.

5.1 Terminology

Notation. Given two ERC20 tokens, X token and Y token, a DEX following the CPMM model for the two tokens is denoted by DEX_{XY}. We use the notation $BAL_X(S, entity)$ to denote the X token balance of $entity$ at blockchain state S. Furthermore, a transaction, tx, is a sequence of calls $c_1c_2c_3$ ... c_n to contracts and are executed atomically in that order. Given initial blockchain state S, the blockchain state after executing transaction tx is S_{tx}.

Profitability. We define profitability as gaining one token type in one transaction. Let $\mathbb{T}$ be the set of all tokens. A transaction tx is profitable with respect to token X if

- $BAL_X(S, attacker) < BAL_X(S_{tx}, attacker)$ and
- $BAL_Y(S, attacker) \leq BAL_Y(S_{tx}, attacker)$ for all $Y \in \mathbb{T} \setminus \{X\}$.

We limit our scope to call sequences that can be executed in one transaction to exclude the impact of interest accumulation and other market players. In a typical attack scenario, X token would be a coin with a relatively stable value, such as the wrapped native currency (e.g., WETH or WBNB) or a stablecoin (e.g., USDT). In addition, for a transaction to be truly profitable, the profit from the transaction should offset the gas fee involved in executing the transaction. However, precise gas fee estimation is difficult because gas fees fluctuate based on several factors, including network congestion. Thus, *for simplicity, we only consider transactions making profits over 1 USD as profitable.*

CPMM composability bugs. Consider a system composed of X token, Y token, and DEX_{XY} following the CPMM model. We *define CPMM composability bug as a bug in the Y token contract that enables an attacker to illegitimately extract X tokens from DEX_{XY} to craft a profitable transaction with respect to the X token.* Here, we assume that the X token contract and the DEX_{XY} contract are free from vulnerabilities. This is a reasonable assumption given that X is a widely used stablecoin and DEX_{XY} follows a standard implementation such as Uniswap.

In general, users cannot gain X tokens by simply interacting with DEX_{XY}; however, if the Y token contract contains a vulnerability or an incompatible behavior, an attacker can extract more than the initially inputted X tokens from DEX_{XY}. Under the assumption that the X token contract and the DEX_{XY} contract are free from vulnerabilities, there are only two ways to extract more X tokens from DEX_{XY}. That is, the attacker can either (1) decrease the Y token balance of DEX_{XY} (i.e., y) or (2) increase Δy. Recall that the formula for calculating the X token output (i.e., Δx) from a CPMM swap is $(x - \Delta x) \times (y + \Delta y) = k$. If y decreases, $k(= x \times y)$ also decreases, increasing Δx. Similarly, when Δy increases, Δx can also be increased. We further categorize CPMM composability bugs into two types based on the invariant broken, as detailed in the following sections.

5.2 Type 1: DEX Token Balance Decrease

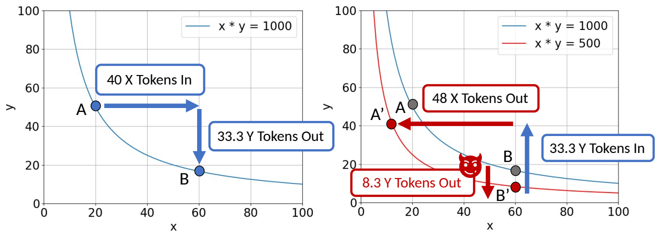

Fig. 5. Example attack scenario where the attacker is able to decrease Y token balance of DEX_{XY}.

```
1 function exploit() public {
2   swapBNBtoShadowFi();
3   // Decrease DEX ShadowFi balance
4   ShadowFi.burn(DEX_ADDR,
          DEX_SHADOWFI_BALANCE-1);
5   Pair(DEX_ADDR).sync();
6   swapShadowFitoBNB();
7 }
```

Fig. 6. Simplified ShadowFi exploit.

The first type of CPMM composability bug is a bug that allows an attacker to decrease the Y token balance of DEX_{XY} without paying. An example scenario is shown in Figure 5. We assume that DEX_{XY} has 20 X tokens and 50 Y tokens with $k = 1000$. The attacker first swaps 40 X tokens for 33.3 Y tokens. Then, the attacker utilizes a CPMM composability bug to decrease y by 8.3 tokens, which decreases k to 500; $60 \times (16.7 - \mathbf{8.3}) \approx \mathbf{500}$. This effectively shifts the swap curve inward. Since the attacker only decreased y and x remains the same, the next swap will happen at a point straight below the previous point on the updated swap curve (i.e., point B′ instead of point B). The price of the Y token is greater at this point, allowing the attacker to swap the same amount of Y tokens for more X tokens (i.e., swapping to point A′ instead of point A). The attacker ends up with 48 X tokens; $(60 - \mathbf{48}) \times (8.4 + 33.3) \approx 500$, which is 8 X token profit. As illustrated with the example, when this invariant is broken, the attacker can extract X tokens from DEX_{XY}.

Invariant 1 (DEX token balance decrease). *Users should not be able to transfer or burn assets owned by a DEX without paying the DEX.*

Real-world example. For instance, on September 2, 2022, an attacker exploited a vulnerability in the ShadowFi token to steal around 1078 BNB (worth around 301K USD) [28]. The vulnerability was that the ShadowFi token had a public burn function. In DeFi, token burning refers to permanently removing tokens from circulation. In the exploit, the public burn function allowed any user to remove ShadowFi tokens owned by any user. The simplified exploit is shown in Figure 6. The attacker decreased the ShadowFi balance of the BNB-ShadowFi DEX using the burn function (line 4) and updated the k value of the DEX (line 5) to swap ShadowFi tokens for more BNB.

5.3 Type 2: Attacker Token Balance Increase

The second type of CPMM composability bug is a bug that allows an attacker to gain Y tokens without cost. An example scenario is shown in Figure 7. At point B, if the attacker can increase its own balance of token Y, then the attacker can gain more than the expected amount of X tokens (i.e., swapping to point A' instead of point A). For the example, the attacker gains an additional 30 Y tokens, resulting in 8 X token profit; $(60 - \mathbf{48}) \times (16.7 + 33.3 + \mathbf{30}) \approx 1000$. Hence, when **Invariant 2** is broken, the attacker can extract X tokens owned by DEX_{XY}.

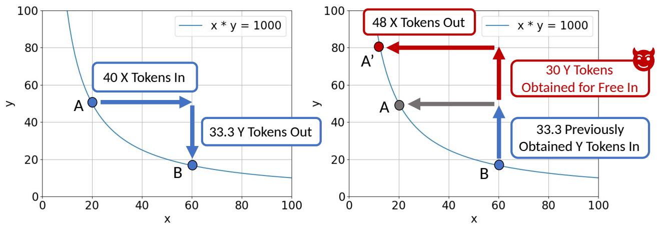

Fig. 7. Example attack scenario where the attacker is able to increase one's own balance of Y tokens.

```
1  function exploit() public {
2    swapBNBtoANCH();
3    AMCH.transfer(DEX_ADDR);
4    // Increase attacker ANCH balance
5    for (uint i; i < 60; ++i) {
6      Pair(DEX_ADDR).skim(DEX_ADDR);
7    }
8    Pair(DEX_ADDR).skim(THIS_ADDR);
9    swapANCHtoBNB();
10 }
```

Fig. 8. Simplified ANCH exploit.

Invariant 2 (Attacker token balance increase). *Users should not be able to obtain tokens traded in a DEX without cost.*

Real-world example. For example, on August 9, 2022, an attacker leveraged ANCH token's reward mechanism to steal around 19.9K BSC-USD [4]. The ANCH contract rewards users who buy or sell ANCH tokens from the ANCH-BSC-USD DEX. However, as shown in lines 4 to 7 in Figure 8, the attacker could illegitimately trigger the reward mechanism by abusing the `skim` function in DEX that makes the DEX send extraneous tokens to any address given as the argument (`skim` function exists to enable DEX to return leftover tokens from swaps to users). Utilizing this mechanism, the attacker could drain BSC-USD from the ANCH-BSC-USD DEX.

6 Design

Based on the formalized model in §5, we design *CPMMX*, an automatic tool that detects CPMM composability bugs. In this section, we describe its design in detail.

6.1 Workflow

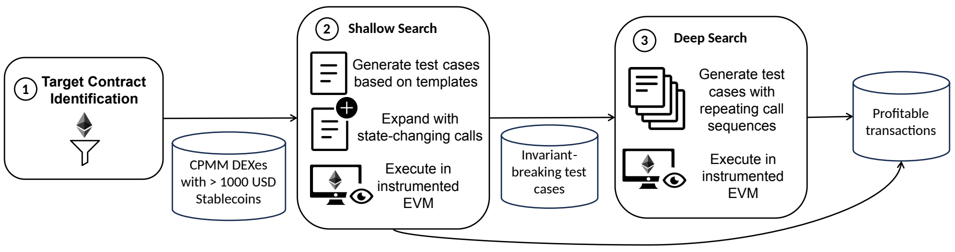
① Target Contract Identification
CPMM DEXes with > 1000 USD Stablecoins
② Shallow Search
Generate test cases based on templates
Expand with state-changing calls
Execute in instrumented EVM
Invariant-breaking test cases
③ Deep Search
Generate test cases with repeating call sequences
Execute in instrumented EVM
Profitable transactions

Fig. 9. Overall workflow of *CPMMX*.

The overall workflow of *CPMMX* is illustrated in Figure 9. ① *CPMMX* first identifies target contracts from the blockchain (§6.2). As *CPMMX* focuses on CPMM composability bugs, it only considers contracts that follow the ERC-20 standard and are traded on a CPMM DEX. Moreover, it is not interesting to analyze contracts with insignificant financial value. Therefore, *CPMMX* filters out DEXes with less than 1,000 USD worth of stablecoins or native currencies. ② Then, *CPMMX* employs a two-phase search strategy, *shallow-then-deep search*, to determine if the target contracts are vulnerable to CPMM composability bugs. The shallow search phase uses predefined templates to find invariant-breaking transactions (§6.3). If such transactions are already profitable, *CPMMX* outputs the transaction and flags the contract as vulnerable. On the other hand, if it cannot find any invariant-breaking transactions, *CPMMX* discards the contract (i.e., early termination). ③ If *CPMMX* can only find invariant-breaking transactions, not profitable ones, it proceeds to the deep search phase (§6.4). In the deep search phase, the invariant-breaking call sequences are repeated to generate a profitable transaction for the target contracts.

Table 2. Templates for generating transactions by *CPMMX* whose bolded elements are repeatable.

Type	Testcase	Description
Cross-trading	**`transfer(this, transferAmount)`**	Send tokens to ourselves
	`transfer(DEX, transferAmount)` **`DEX.skim(this)`**	Send tokens back and forth between DEX and ours
	`transfer(DEX, transferAmount)` **`DEX.skim(DEX)`** `pair.skim(this)`	Send tokens to DEX, DEX sends tokens to itself, DEX sends tokens to ours
	`transfer(DEX, transferAmount)` **`DEX.skim(DEX)`**	Send tokens to DEX, DEX sends tokens to itself
Burn	**`burn(burnAmount)`**	Remove tokens from circulation
	`burn(DEX, burnAmount)`	Remove tokens from DEX

6.2 Finding Target Contracts from the Blockchain

First, *CPMMX* scans the blockchain to find target contracts. Our targets are token contracts traded in a CPMM DEX with a meaningful financial value. To this end, *CPMMX* retrieves the addresses of all DEX contracts from the UniswapV2 factory contracts on both the Binance Smart Chain (BSC) and Ethereum networks. We used the most popular CPMM DEX platforms in each blockchain: PancakeSwap for BSC and UniswapV2 for Ethereum. Then, for each DEX contract, *CPMMX* collects the addresses of the tokens traded within the DEX and the balance of each token. From the list of all DEX contracts, *CPMMX* filters out contracts that do not meet the following criteria.

- **Exchangable to standard tokens**. One of the tokens traded in the DEX is the wrapped native currency or a well-known stablecoin.
- **Financially meaningful**. The DEX contains over 1,000 USD worth of the wrapped native currency or a well-known stablecoin.

The DEX contracts that meet the above criteria and the associated tokens are then considered target contracts. As of October 2024, there were 1,674,491 PancakeSwap contracts and 371,380 UniswapV2 contracts. After filtering, we had 19,377 (1.16%) and 28,800 (7.74%) contracts, respectively.

6.3 Shallow Search for Finding Invariant-Breaking Transactions

Overview. After identifying target contracts, *CPMMX* begins the shallow search to find invariant-breaking transactions. This works as follows. First, *CPMMX* deploys and initializes the attacker contract with a stablecoin balance. Second, it generates a transaction consisting of calls $c_1 c_2 c_3 \ldots c_n$ where c_1 and c_n are calls to swap the attacker contract's stablecoins to the target token via the victim DEX and vice versa, and calls $c_2 \ldots c_{n-1}$ are generated based on predefined templates. Then, *CPMMX* executes the calls in an instrumented EVM environment and determines whether the transaction is profitable or breaks any invariants. Finally, if no profit-generating transactions are found, *CPMMX* expands the test cases by incorporating state-changing calls.

Generating test cases with invariant-breaking templates. To build transactions likely to break invariants, *CPMMX* uses templates, which are described in Table 2, to generate transactions. These templates have been inspired by patterns observed in previous exploits [7, 12]. From our analysis of these exploits, we observed that CPMM composability bugs are typically caused by incentive mechanisms embedded in tokens. This is quite an intuitive observation because most tokens are designed to incentivize users to promote their tokens. Some tokens incentivize trading by rewarding users on specific transfers or by employing deflationary mechanisms, removing tokens from circulation to increase their value.

To test incentive mechanisms of tokens, *CPMMX* uses two types of templates: cross-trading and token burning. First, *CPMMX* uses cross-trading templates to execute token transfers without causing a net change in balances. As shown in Table 2, the first three strategies for cross-trading

Table 3. Arguments used in the shallow search phase.

Parameter	Argument Values

transferAmount	`token.balanceOf(this)`
	`token.balanceOf(pair)`
	`0`

burnAmount	`token.balanceOf(pair) - 1` `token.totalSupply() - 2 * token.totalSupply() \ / token.balanceOf(pair)` [31]

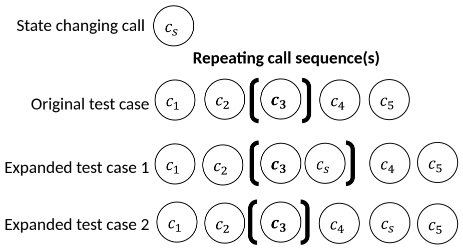
State changing call
c_s
Repeating call sequence(s)
Original test case
c_1 c_2 c_3 c_4 c_5
Expanded test case 1
c_1 c_2 c_3 c_s c_4 c_5
Expanded test case 2
c_1 c_2 c_3 c_4 c_s c_5

Fig. 10. Incorporating a state changing call to a template testcase.

are trivial to see why they are cross-trading. The fourth one is a bit more subtle, as the tokens are left in the DEX. However, since *CPMMX* immediately swaps all the tokens afterward (c_n), the remaining tokens in DEX are treated as inputs for the swap. We also included the self-transfer template (i.e., the first row of Table 2). Although it does not directly interact with the victim DEX, it can still trigger invariant-breaking behaviors leading to CPMM composability bugs and stablecoin losses. Second, *CPMMX* also attempts to test token-burning functions. As mentioned, some tokens include explicit burn functions designed to remove tokens from circulation. To test this feature, *CPMMX* includes any function containing the term "burn" in its name. For argument values, we populate them with values commonly observed in exploits. The values are listed in Table 3.

Our approach is similar to other template-based searches, but unlike others, our templates have special, *repeatable* elements. In Table 2, the repeatable elements are highlighted in bold. These elements involve the specific mechanisms that are likely to break invariants (e.g., transferring vulnerable tokens to/from DEX, burning tokens), and repeating them does not disrupt the cross-trading nature of the test cases. With these repeatable elements, we divide our search into two phases: shallow search and deep search. In the shallow search, *CPMMX* uses the templates without repeating them, enabling a quick assessment to discard contracts that do not break invariants. The deep search repeats the critical call sequences to potentially generate a profitable transaction. This design is crucial, as invariant-breaking sequences often need multiple executions to offset other costs and ultimately yield profit. However, incorporating repetition from the shallow search phase would unnecessarily increase the time needed to discard benign contracts.

Executing test cases in instrumented EVM. After generating a test case, *CPMMX* runs it in an instrumented EVM to check whether the test case breaks any invariants from the formalized model in §5. For that, *CPMMX* uses the instrumented EVM to track important state variables, such as the attacker's and DEX's token balances. It also keeps a snapshot of the state variables before executing c_2. It compares them with the state variables after executing until c_{n-1} to determine whether the call sequence c_2 ... c_{n-1} breaks any invariants. Finally, by comparing the amount of stablecoins before and after the transaction, *CPMMX* can determine whether the transaction is profitable.

Expanding test cases with state-changing calls. If no profit-generating transactions are found, *CPMMX* expands templates by incorporating state-changing calls. It considers two types of state-changing calls: small-amount transfers and no-argument function calls. Similar to the templates, this approach is inspired by token incentive mechanisms. Some tokens include token price maintenance logic in transfer functions. Small amount transfers can trigger these functions without significantly impacting token balances. Thus, *CPMMX* includes small amount transfers (i.e., transfers with amount 0 or 1) in state-changing calls. On the other hand, some tokens have dedicated functions to trigger incentive mechanisms. While these functions may accept arguments, *CPMMX* only considers no-argument function calls. This design choice is based on two reasons. First, generating valid arguments for function calls is a nontrivial task that can greatly reduce efficiency. Second, restricting *CPMMX* to only no-argument function calls still detects most exploits. Hence, *CPMMX* includes only no-argument function calls in state-changing calls. Figure 10 illustrates how a state-changing call is incorporated into a template test case. Each template is combined with a state-changing call to create two additional test cases. In the first, the state-changing call is appended at the end of the repeating call sequence. In the second, it is inserted just before the last swap. The state-changing calls are deliberately inserted in these positions to preserve the self-trading nature of the test cases. These test cases are then executed and checked for broken invariants or profit generation. If no such transactions are detected, *CPMMX* terminates early. Our approach is effective in practice; however, it is not exhaustive. This limitation is discussed in §10.2.

6.4 Deep Search for Generating Profitable Transactions

As explained in §3.1, an invariant-breaking transaction is insufficient to assure a contract is vulnerable. This is because many benign tokens can exhibit similar behaviors but cannot be exploited to yield profit. Thus, to eliminate such false positives, *CPMMX* employs the deep search phase to craft a proof of vulnerability (i.e., a profitable transaction).

Executing test cases with repetitions. If the shallow search phase identifies only invariant-breaking transactions, *CPMMX* attempts to generate a profitable transaction by exacerbating the invariant violation through repeating invariant-breaking call sequences. Thus, in the deep search phase, it generates test cases with increasing repetitions. Among these, we need to allocate more resources to test cases that are more likely to yield profit. To that end, *CPMMX* utilizes the final stablecoin balance after executing a test case as a guide to decide which test cases to prioritize. If the final balance remains unchanged after increasing repetitions three times, it dismisses the test case, assuming further repetitions will not change the contract state. If the final balance decreases, the test case is retained but with limited repetitions, as profitability is unlikely. If the final balance increases, it continues increasing repetitions but still enforces a cap to avoid wasting resources on slow-growing profits.

7 Implementation

CPMMX was built on top of Foundry [14], a well-known smart contract testing tool, and relies on it to set up the environment necessary for on-chain testing. Furthermore, *CPMMX* fetches contract ABIs from popular blockchain explorers: BSCScan [1] and Etherscan [2].

State tracking and argument replacement. One challenge in testing smart contracts is generating test cases that do not revert. For instance, in a typical `transfer(address, amount)` function, a transaction reverts if `amount` exceeds the sender's balance. Therefore, *CPMMX* must generate a value less than the sender's balance. Moreover, if the token transfer involves an exclusive fee, the transfer amount must account for enough balance to cover the fee to avoid transaction reverting. Generating such valid arguments is challenging as token balances may change after each call execution. To address this issue, *CPMMX* leverages state tracking and runtime argument replacement. It monitors important state variables, such as token balances, and replaces arguments with the most current values during execution. Furthermore, when exclusive fees apply, it computes the appropriate transfer amount to ensure that the transaction proceeds without reverting.

8 Evaluation

To evaluate *CPMMX*, we answer the following research questions:

- **RQ1:** How effective is *CPMMX* at detecting CPMM composability bugs?
- **RQ2:** How efficient is *CPMMX* at detecting CPMM composability bugs?
- **RQ3:** How significant are the techniques applied to *CPMMX*?
- **RQ4:** How effective is *CPMMX* at detecting undiscovered CPMM composability bugs in the real world?

8.1 Experimental Setup

8.1.1 Baseline Selection. Among many existing tools, we selected five tools as baselines: ItyFuzz [29], Echidna [17], DeFiTainter [21], Slither [13] and Mythril [27]. We selected ItyFuzz, Echidna, and DeFiTainter as they support multi-contract analysis and can detect (a subset of) CPMM composability bugs. We also included Slither and Mythril, which do not support multi-contract analysis, to demonstrate that tools designed for single contracts are ineffective at detecting CPMM composability bugs.

In the following, we describe the configurations for each tool used in the evaluation. Note that we tried our best to configure each tool for fair comparison. Furthermore, DeFiTainter and Slither require source code analysis, so we could not run them for close-sourced contracts.

- **ItyFuzz**. We ran ItyFuzz with only the bug oracle that detects ERC20 token leaks. ItyFuzz also has a bug oracle for detecting token imbalances in DEXes (i.e., violations of **Invariant 1**), but these issues do not always lead to vulnerabilities. Therefore, we set up ItyFuzz to detect profitable transactions, similar to how *CPMMX* operates as a whole.
- **Echidna**. Echidna requires custom oracles to detect vulnerabilities. Thus, we implemented an oracle that checks whether the attacker contract can get more native tokens after exchanging all ERC20 tokens for native ones.
- **DeFiTainter**. DeFiTainter determines whether a given function contains a price manipulation vulnerability. Thus, we ran DeFiTainter for all public and external functions of a target contract and flagged it as vulnerable if any of the functions outputted a positive result.
- **Mythril**. Mythril has no detector for ERC20 token or ether leaks. However, other detectors might detect the programmatic error, leading to broken safety invariants for CPMMs. Thus, we manually validated results to check if Mythril can find the root cause of each exploit.
- **Slither**. Since Slither includes many non-critical detectors, we ran Slither with only detectors that could be a potential root cause for CPMM composability bugs (i.e. arbitrary-send-erc20, protected-vars, arbitrary-send-erc20-permit, arbitrary-send-eth, unchecked-transfer). Then, we manually validated its result to check if it could discover the root cause of each exploit.

8.1.2 Datasets. We used three datasets for comparing *CPMMX* with existing tools: two public exploit datasets (DeFiHackLabs and BlockSec) and one custom-built dataset for evaluation (RealWorld-BSC). We use these datasets to answer **RQ1**, **RQ2**, and **RQ3**.

- **DeFiHackLabs (N = 23)**. First, we use DeFiHackLabs [12], a public dataset for DeFi hacking incidents. This dataset is widely used for evaluating smart contract analysis tools [30, 33, 35]. We used 23 exploits from this dataset that utilize CPMM composability bugs as shown in 3.2.
- **BlockSec (N = 124)**. Second, we also used BlockSec [6], a public dataset containing 138 real-world exploits that involve breaking **Invariant 1**. Out of the 138 exploits, we use 124 exploits for this evaluation, as 14 are duplicate ones. Only one exploit, the SHEEP token exploit, is included in both the DeFiHackLabs and BlockSec datasets.
- **RealWorld-BSC (N = 244)**. Third, we constructed a dataset named RealWorld-BSC. Unlike other datasets, we attempt to include both vulnerable and benign contracts to measure the real-world performance of each tool. This dataset consists of 122 vulnerable contracts from BlockSec, which are deployed on the BSC, and the same number of benign contracts. We use early-deployed DEX contracts with over 100 WBNB (worth around 60,000 USD) from PancakeSwap for benign contracts. This is based on our assumption that tokens traded for a long time and holding substantial assets are less likely to be vulnerable.

In addition to these datasets, *CPMMX* was also run on a large scale ($N > 10,000$) on the Ethereum and Binance chains to evaluate its effectiveness in the real world (**RQ4**).

8.1.3 Timeout and Number of Trials. We used 20 minutes as the timeout for each contract, which is reasonably long enough if we consider the number of contracts to analyze in the real world. We ran fuzzing-based approaches (i.e., *CPMMX*, ItyFuzz, and Echidna) three times for each case and reported the average values to avoid non-deterministic results from fuzzing.

8.2 Effectiveness in Detecting Composability Bugs

Table 4. CPMM composability bug detection rate of *CPMMX* and baselines on the DeFiHackLabs dataset.

	AES	ANCH	ApeDAO	Bamboo	BFC	BGLD	BIGFI	BRA	GHT	GPT	HCT	HEALTH	LPC	PLTD	pSeudoEth	Shadowfi	SHEEP	Starlink	TGBS	ThoreumFi	Upswing	Wdoge	XST	Total	Recall
DeFiTainter	0	0	1	0	0	0	0	0	-	-	0	0	0	0	-	0	0	0	0	-	0	0	0	1/19	0.05
Echidna	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0/23	0.00
ItyFuzz	1	0	0	1	0.33	0	0.33	0	0	0	0.33	0	1	0	1	0	0.33	0	1	0	1	1	0	8.33/23	0.36
Mythril	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0/23	0.00
Slither	0	0	0	0	0	0	0	0	-	-	0	0	0	0	-	0	0	0	0	-	0	0	0	0/19	0.00
Ours	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	21/23	**0.91**

Table 5. CPMM composability bug detection rate of *CPMMX* and baselines on the BlockSec dataset.

	DeFiTainter	Echidna	ItyFuzz	Mythril	Slither	Ours
Total	1/123	9/124	74/124	0/123	0/124	**109/124**
Recall	0.01	0.07	0.60	0.00	0.00	**0.88**

To compare the effectiveness of *CPMMX* in detecting CPMM composability bugs with existing tools, we measured the recall of *CPMMX* and each baseline. Table 4 and Table 5 show the results of running *CPMMX* and baselines on the DeFiHackLabs and BlockSec datasets, respectively. Note that we report the average detection rates for fuzzers, which may result in fractional values.

In summary, *CPMMX* outperformed other tools in detecting CPMM composability bugs. In DeFiHackLabs dataset (Table 4), *CPMMX* achieved the highest recall value of 0.91, while ItyFuzz had the second-highest recall value of 0.36. DeFiTainter detected only one vulnerability out of 19 contracts, thus having a recall value of 0.05. Other tools failed to detect any vulnerabilities. In the BlockSec dataset (Table 5), *CPMMX* also achieved the highest recall of 0.88, while ItyFuzz had the second-highest recall of 0.60.

CPMMX achieved significantly higher recalls than other tools by efficiently targeting areas of the search space likely to contain profitable exploits for CPMM composability bugs. Some exploits, such as ANCH, require repeated invariant-breaking call sequences to yield profit. These repetitions do not increase code coverage but gradually change contract states. Such scenarios are unlikely to be explored by coverage-guided fuzzers like Echidna and ItyFuzz, whereas *CPMMX* addresses this search space effectively through its deep search phase. DeFiTainter was ineffective at detecting CPMM composability bugs; it detects a different type of vulnerability — price manipulation vulnerabilities — and can only cover a subset of CPMM composability bugs. Similarly, it is unsurprising that Mythril and Slither could not detect any CPMM composability bugs. Mythril and Slither are static analyzers limited to analyzing individual contracts, whereas CPMM composability bugs arise from the interaction between vulnerable tokens and DEX contracts. The results indicate the need for a targeted approach to detect CPMM composability bugs.

CPMMX could not exploit ApeDAO and GHT because of their unique characteristics unlike other contracts. In particular, ApeDAO's fee mechanism is special as it is cheaper to pay fees in stablecoins rather than in ApeDAO tokens during DEX exchanges. As a result, a calculated stablecoin transfer to the DEX is required to exploit this behavior, which is not covered by *CPMMX*.

For GHT, attackers exploited the vulnerability in the same block where developers deposited GHT into the DEX. Since *CPMMX* runs on finalized state variables and does not account for intra-block transactions, all its test cases reverted due to the absence of GHT tokens in the victim DEX. This limitation is further discussed in §10.2.

Answer to RQ1: *CPMMX* outperforms existing tools in detecting CPMM composability bugs.

8.3 Efficiency in Detecting CPMM Composability Bugs

Table 6. Performance metrics and running time comparison of *CPMMX* and baselines on the RealWorld-BSC dataset at block 25543755.

	Echidna	ItyFuzz	Ours
Precision	0.80	**1.00**	**1.00**
Recall	0.09	0.49	**0.93**
F1 Score	0.16	0.66	**0.97**
Vulnerable Time (min)	2290	1707	**150**
Benign Time (min)	2425	2440	**447**
Overall Time (min)	4715	4147	**597**
Vulnerable Timeout #	111.33	62	**6**
Benign Timeout #	119.33	122	**16**
Overall Timeout #	230.66	184	**22**

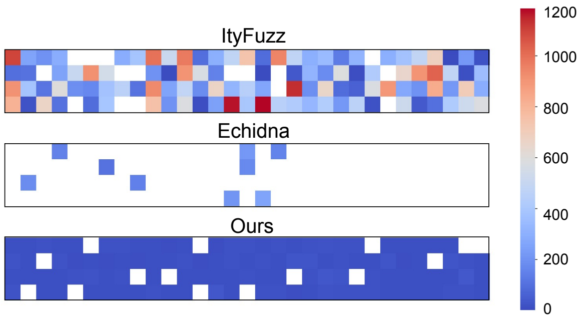

Fig. 11. Heatmap of time taken to detect CPMM composability bugs in BlockSec dataset (seconds).

8.3.1 Precision. To evaluate the precision of *CPMMX* against other tools, we tested *CPMMX*, ItyFuzz and Echidna on the RealWorld-BSC dataset. Other tools were excluded because they could not detect any vulnerabilities in the BlockSec dataset. To simulate a scenario where these tools scan the entire blockchain for vulnerabilities and to ensure consistent contract states, we ran all experiments at block 25543755, which is one block before the earliest exploit in the BlockSec dataset.

Table 6 presents the results of running *CPMMX* and baselines on the RealWorld-BSC dataset. For precision, *CPMMX* and ItyFuzz achieved the highest precision with 1.00. Echidna reported a few false positive cases. We reviewed these cases and found that Echidna incorrectly flagged reverting transactions as invariant violations. Specifically, when converting tokens back to native currencies for comparison, some transactions reverted, leading Echidna to incorrectly classify them as invariant violations. Since such transactions would not be executed in real-world scenarios, we classified them as false positives. Although ItyFuzz did not report false positives, it can identify much fewer CPMM composability bugs compared to *CPMMX*. Please note that the recall values differ from those in Table 5. This is because in the previous evaluation, we used different block numbers for each contract to ensure that the contracts are exploitable (i.e., one block before the contracts were exploited); however, in this evaluation, we used a single block number, 25543755.

8.3.2 Time. To evaluate the efficiency of *CPMMX* in detecting CPMM composability bugs, we compare its running time to that of ItyFuzz and Echidna. Table 6 shows the average running time of each tool on the RealWorld-BSC dataset. *CPMMX* was the most efficient, taking 597 minutes (around 10 hours) to test all 244 contracts. In comparison, Echidna and ItyFuzz took 4147 minutes (around 69 hours) and 4715 minutes (around 79 hours), respectively. The efficiency of *CPMMX* can be largely attributed to its ability to terminate early for contracts that do not exhibit invariant violations, resulting in only 16 out of 122 benign contracts reaching the timeout limit. In contrast, other tools run until timeout if they cannot detect any vulnerabilities. If the timeout had been extended, the running times of Echidna and ItyFuzz would have likely been even longer. In addition, in §A.1, we evaluate *CPMMX*'s performance with benign fee-on-transfer tokens, which are more challenging to differentiate from tokens with CPMM composability bugs.

Table 7. Average time taken by *CPMMX* and ItyFuzz to detect bugs in the DeFiHackLabs dataset (seconds).

	AES	ANCH	ApeDAO	Bamboo	BFC	BGLD	BIGFI	BRA	GHT	GPT	HCT	HEALTH	LPC	PLTD	pSeudoEth	Shadowfi	SHEEP	Starlink	TGBS	ThoreumFi	Upswing	Wdoge	XST	Average
ItyFuzz	508	-	-	17	86	-	927	-	-	-	177	-	102	-	26	-	795	-	**15**	-	33	**27**	-	246
Ours	**41**	**38**	-	**12**	**61**	**15**	**31**	**48**	-	**19**	**30**	**54**	**22**	**65**	**7**	**299**	**11**	**113**	86	**24**	**10**	75	**11**	**51**

CPMMX also detects CPMM composability bugs much faster than other tools. Table 7 shows the average time taken to detect each vulnerability in the DeFiHackLabs dataset. The last column shows the average time taken to detect vulnerabilities. On average, *CPMMX* took 51 seconds to detect vulnerabilities, around 4.82 times faster than the average time taken by ItyFuzz. Moreover, Figure 11 is a visual representation of how fast each tool was at finding each vulnerability in the BlockSec dataset. Each cell in the heatmap contains the result for one vulnerability, thus a total of 124 cells per tool. The red color indicates that the tool took a long time (close to 1,200 seconds or 20 minutes) to detect the vulnerability, while the blue color indicates that the tool took a short time (close to 0 seconds) to detect the vulnerability. White cells indicate that the tool could not detect vulnerability for all three trials. As shown in the figure, *CPMMX* could detect most vulnerabilities quickly, while ItyFuzz detected vulnerabilities in varying time frames. On average, ItyFuzz and Echidna took 383 seconds and 178 seconds to detect vulnerabilities, respectively. Meanwhile, *CPMMX* took only 16 seconds to detect vulnerabilities, around 24 times faster than the average time taken by ItyFuzz and 11 times faster than the average time taken by Echidna.

This evaluation demonstrates that *CPMMX* effectively detects CPMM composability bugs at scale. However, it does not establish that *CPMMX* is more scalable overall because ItyFuzz and Echidna are designed for comprehensive smart contract testing rather than for rapid identification of specific vulnerabilities (e.g., CPMM composability bugs). Thus, they do not prioritize minimizing execution time. The results instead suggest that a targeted approach, which efficiently identifies the existence of specific bugs, is more suitable for large-scale vulnerability detection.

Answer to RQ2: *CPMMX* is the most efficient tool to detect CPMM composability bugs. It requires the least running time and does not report any false positives.

8.4 Ablation Study

To assess the design of *CPMMX*, we perform ablation studies to measure the impact of shallow-then-deep search (§8.4.1) and runtime argument replacement (§8.4.2). Furthermore, in §A.2, we present an additional ablation study to compare *CPMMX* with a commonly used auditing technique that modifies DEX contracts to remove fees.

8.4.1 Shallow-Then-Deep Search. To demonstrate the effectiveness of our approach, we conducted ablation studies to evaluate its impact on bug detection rate and measure its instruction coverage. **Bug detection**. We conducted an ablation study with two modified versions of *CPMMX*: *CPMMX-NoRepeat* and *CPMMX-NoInvariant*. *CPMMX-NoRepeat* does not utilize repetitions for test case generation (i.e., only shallow search). *CPMMX-NoInvariant* generates test cases with a random number of repetitions and directly checks for profit generation (i.e., only deep search). Similar to previous evaluations, we ran *CPMMX-NoRepeat* and *CPMMX-NoInvariant* three times.

On average, *CPMMX-NoRepeat* detected 11 and *CPMMX-NoInvariant* detected 16 out of 23 vulnerabilities in the DeFiHackLabs dataset, which is significantly lower than the 21 vulnerabilities detected by *CPMMX*. Such an outcome is expected; *CPMMX-NoRepeat* cannot detect vulnerabilities that require repetition for profit. Meanwhile, *CPMMX-NoInvariant* cannot efficiently allocate resources to function calls that are more likely to lead to exploits.

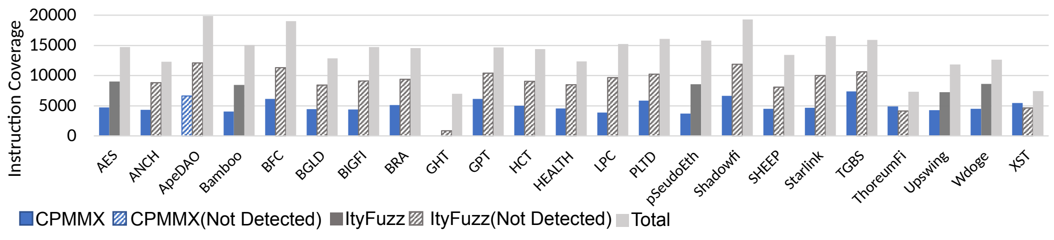

Fig. 12. Code coverage reached by *CPMMX* and ItyFuzz on DeFiHackLabs dataset.

Instruction coverage. We also compare the instruction coverage of *CPMMX* with that of ItyFuzz. Total instruction coverage is measured as the sum of all instructions in the DEX contract and the corresponding token contracts. The results, presented in Figure 12, reveal that while *CPMMX* detects more CPMM composability bugs, ItyFuzz generally achieves higher instruction coverage. This suggests that ItyFuzz explores a broader range of functions than *CPMMX*, as it is designed to cover diverse vulnerabilities. In contrast, *CPMMX* is specially designed to detect CPMM composability bugs. Its shallow search phase effectively filters out irrelevant functions, enabling a more targeted exploration of functions likely to reveal these vulnerabilities.

8.4.2 Runtime Argument Replacement. To illustrate the necessity of *runtime argument replacement*, we compare *CPMMX* to *CPMMX-NoArgReplace*, which does not employ this technique. More specifically, *CPMMX-NoArgReplace* stops state tracking after the initial swap to the target token to execute as if *CPMMX* generated test cases with fixed arguments. In the DeFiHackLabs dataset, *CPMMX-NoArgReplace* detected only 5 out of 23 vulnerabilities. This is because it is highly prone to generating test cases that revert, as it cannot account for changing token balances. Furthermore, *CPMMX-NoArgReplace* cannot detect invariant violations that occur midway through a test case.

Answer to RQ3: The design of *CPMMX*, particularly shallow-then-deep search and runtime argument replacement, play a critical role in effectively detecting CPMM composability bugs.

8.5 Effectiveness in the Real World

To demonstrate the effectiveness of *CPMMX* in detecting undiscovered CPMM composability bugs in the real world, we ran *CPMMX* on the latest blocks of Ethereum and Binance. Table 8 contains the summary of profitable transactions generated by *CPMMX*. *Please note that we represent them with exploit numbers instead of token names or addresses because these vulnerabilities have not yet been patched.* We discuss the issues with responsible disclosure for smart contracts in §10.1.

In summary, *CPMMX* could generate 26 profitable transactions by exploiting CPMM composability bugs in the real world, resulting in a total 15.7K USD profit. To demonstrate the impact of each vulnerability, we report the maximum achievable profit in USD (column 4) and the ratio of this profit to the pair's balance before the exploit (column 5). As *CPMMX* halts when it finds a profit-generating transaction and does not proceed to maximize profit, we manually adjusted some parameters of the exploit (e.g., initial token balance or the number of repetitions) to maximize the profit. According to our experience, this profit maximization was straightforward for all exploits.

Table 8. Real-world exploits generated by *CPMMX*. As these vulnerabilities have not been patched, we denote them with numbers to avoid providing details for exploitable vulnerabilities.

Exploit Number	Invariant Broken	Nework	Max Profit in USD	% Pair Asset
1	1	BSC	213.60	1.68
2	2	BSC	398.00	0.74
3	1	BSC	125.00	2.86
4	1	BSC	**4796.00**	**189.66**
5	1	BSC	282.76	2.40
6	1	BSC	76.64	3.54
7	1	BSC	2.14	0.05
8	1	BSC	1.77	0.19
9	1	BSC	1.80	0.25
10	1	BSC	3.70	0.16
11	1	BSC	1.43	0.0046
12	2	BSC	614.00	1.14
13	2	ETH	148.77	0.24
14	1	ETH	263.61	1.66
15	2	ETH	23.49	0.28
16	1	ETH	20.88	0.39
17	1	ETH	**4358.70**	**99.85**
18	1	ETH	13.05	1.90
19	2	ETH	631.62	**56.93**
20	1	ETH	46.98	0.71
21	1	ETH	4.65	0.17
22	2	ETH	5.90	0.44
23	1	ETH	3.52	0.27
24	1	ETH	21.82	1.81
25	1	ETH	39.15	3.33
26	1	ETH	**3575.70**	**99.77**

CPMMX could generate four critical exploits that can drain more than half of the DEX's stablecoin balance (marked in bold in Table 8). Other attacks can also yield considerable profits (e.g., a few hundred dollars). We currently only consider single transaction exploits. However, if this vulnerability is exploited repeatedly in the long term, it could result in sustained profit and potentially drain all funds. We leave the development of such long-term exploit generation as future work.

Answer to RQ4: *CPMMX* can generate impactful real-world exploits.

9 Case Study

This section reports case studies for real-world CPMM composability bugs that *CPMMX* detected.

```
1 function transfer ( address addr , uint
      amount ) external {
2   if ( addr == DEX_ADDR ) {
3     maintainPrice ();
4   }
5   // transfer tokens
6   balances [ msg . sender ] -= amount ;
7   balances [ addr ] += amount ;
8 }
9 function maintainPrice () internal {
10   // decrease pair token balance by 10%
11   balances [ DEX_ADDR ] =
12   balances [ DEX_ADDR ] * 9 / 10;
13 }
```

Fig. 13. Vulnerable code snippet from Exploit 17.

```
function getRate () public {
  return totalTokenSupply / totalShareSupply ;
}
function transfer ( address addr , uint amount ) external {
  // transfer tokens
  uint shareAmount = amount / getRate ();
  shareBalances [ msg . sender ] -= shareAmount ;
  shareBalances [ addr ] += shareAmount ;
}
function maintainToken () external {
  // check that caller is a token owner
  require ( shareBalances [ msg . sender ] > minAmount );
  // proportionally decrease variables
  totalTokenSupply = totalTokenSupply * 9 / 10;
  totalShareSupply = totalShareSupply * 9 / 10;
  // award caller
  shareBalances [ msg . sender ] += awareAmount ;
}
```

Fig. 14. Vulnerable code snippet from Exploit 19.

9.1 Exploit 17: Breaking Invariant 1

Exploit 17 is a real-world bug that violates **Invariant 1**. This happens due to Token 17's deflationary mechanism. Figure 13 shows the simplified version of vulnerable code from Token 17. According to the CPMM model, whenever users sell Token 17 to the DEX, the price of Token 17 falls. To mitigate the price fall, Token 17 has a function that burns its share in the DEX whenever users sell Token 17 to the DEX (i.e., `maintainPrice()` function in lines 9-13). Since this behavior can be triggered multiple times, an attacker can burn a significant portion of the DEX token balance, breaking **Invariant 1**. The attacker can leverage this vulnerability to drain almost all stablecoins from the

DEX, which is worth 4358.70 USD. Interestingly, it is not profitable if we trigger this behavior only once, making it difficult for existing tools to detect this bug. On the other hand, *CPMMX* can detect this bug by repeatedly triggering the behavior thanks to our two-step approach.

9.2 Exploit 19: Breaking Invariant 2

Exploit 19 is a real-world bug that breaks **Invariant 2**. This token, henceforth Token 19, rewards users whenever they call a maintenance function. Figure 14 shows the simplified version of Token 19. This token manages its balances using two variables, `totalTokenSupply` and `totalShareSupply`. As more users join the market for Token 19, the two variables will increase and may result in integer overflow. To prevent such a situation, Token 19 has to decrease the two variables periodically. Such maintenance function is implemented in lines 10 to 18 in Figure 14. Unlike other tokens that embed these functions in a commonly called function, such as `transfer`, Token 19 adopts a different approach, where users are incentivized to directly call the function with rewards. However, the developers did not limit the number of times a user can call the function. Thus, an attacker can repeatedly call `maintainToken()` to accumulate a significant amount of Token 19 without cost. Note that, similar to our motivating example in §3.1, the attacker has to reap the reward multiple times to offset costs, making it unlikely for existing tools to detect this bug. We concluded that this vulnerability could be leveraged to drain around 57% of relevant DEX's stablecoin balance, which is worth 631.62 USD.

10 Discussion

10.1 Responsible Disclosure for Smart Contract Vulnerabilities

We attempted to notify the token maintainers about bugs found by *CPMMX* but were unable to reach them. We also reported our findings to CISA (Cybersecurity & Infrastructure Security Agency), who recommended public disclosure of the bugs. However, we decided not to proceed with this recommendation, as anyone exploiting the vulnerabilities would directly cause financial harm to token holders. In addition, we consulted with SEAL 911, a group of blockchain security researchers, but could not determine a safe and ethical approach for managing these vulnerabilities. As a result, the vulnerabilities remain unpatched. We hope that a safe and ethical approach will be established for addressing security issues in projects without active maintainers.

10.2 Limitations and Future Works

CPMMX has three limitations. First, it currently only supports Uniswap V2 DEXes. With further development, *CPMMX* can be extended to support other CPMM implementations, which we leave for future work. Second, *CPMMX* is constrained by the templates and state-changing calls used in test case generation. It fails to detect CPMM composability bugs that do not conform to the templates, such as ApeDAO (§8.2). Furthermore, it does not leverage all available functions from token contracts for state-changing calls. It discards function calls with arguments to avoid the complexity coming from generating valid arguments. To mitigate this limitation, we could extend templates and incorporate more state-changing function calls. Static analysis could help generate valid arguments. However, supporting a broader range of templates and functions would also increase the search space, potentially reducing efficiency. Investigating the trade-off between efficiency and generalizability would be valuable in identifying the optimal balance. Finally, *CPMMX* only supports CPMM composability bugs, which can be represented by breaking two safety invariants. In the future, it would be interesting to design a targeted approach like *CPMMX* for other vulnerabilities.

10.3 Threats to Validity

Internal threats. Since we suggest a new category of vulnerability, we could not evaluate our system on well-established datasets. Instead, we selected a subset from a popular dataset, DeFi-HackLabs [12], as one of our evaluation datasets. To mitigate potential internal threats, the first and second authors independently selected the subset and discussed each exploit until reaching a consensus, minimizing selection bias and ensuring a consistent evaluation process.
External threats. A potential external threat is the limited number of reported CPMM composability bugs. To address this, we conducted an in-the-wild experiment as described in §8.5, which allowed us to validate *CPMMX*'s performance against real-world contracts.

11 Related Work

Numerous tools detect smart contract vulnerabilities. Some utilize static analysis techniques, such as model checking [5] and symbolic execution [15, 25]. While others utilize dynamic analysis techniques, most notably fuzzing [11, 17, 29]. Recent works also utilize machine learning [8, 24], including Large Language Models [10, 30].
Multi-contract vulnerability detection. Several works were proposed to detect multi-contract vulnerabilities. Some focus on detecting commonly appearing ones, such as reentrancy and delegatecall-related vulnerabilities [23, 32], while some aim to detect a wide variety of vulnerabilities [17, 29]. In particular, ItyFuzz explores various combinations of contract states through fuzzing with snapshots, and Echidna uses a static analyzer, Slither, to extract useful information before fuzzing. Although CPMM composability bugs can theoretically be detected with such methods, our evaluation indicates that a generic approach is ineffective. Since CPMM composability bugs are closely tied to the business logic of contracts and sometimes require a long sequence of function calls for exploitation, a targeted approach is more suitable, as demonstrated by *CPMMX*.
Automatic exploit generation. Some works propose systems that automatically generate exploits. EthPloit [34] generates exploits for single contracts based on fuzzing. FlashSyn [9] utilizes counterexample-driven approximation to generate flashloan attacks. Gritti et al. [18] designed a system that analyzes multiple contracts to automatically detect and exploit confused deputy vulnerabilities. *CPMMX* pursues the same goal of exploit generation, but it targets a vulnerability that the aforementioned tools cannot detect.

12 Conclusion

We proposed *CPMMX*, a tool that can automatically detect and generate end-to-end exploits for CPMM composability bugs. For this, we first formalized CPMM composability bugs and propose a two-step approach, called shallow-then-deep search, to detect them. We evaluated *CPMMX* against five baselines on three datasets and demonstrated that *CPMMX* outperforms all baselines in terms of recall, precision, and F1 score. Furthermore, we applied *CPMMX* on the latest blocks of the Ethereum and Binance chains and discovered 26 new exploits that can yield 15.7K USD total profit.

13 Data Availability

In support of the open science policy, we make the source code of *CPMMX*, datasets, and scripts to run experiments available at a public repository [19, 20].

Acknowledgments

This work was supported in part by the Korea-U.S. Joint Research Support Program funded by the Ministry of Science and ICT (MSIT) through the National Research Foundation of Korea (NRF) (RS-2022-NR119707) and the NRF grant funded by the Korea government (MSIT) (RS-2024-00337007).

A Appendix

Table 9. Performance metrics and running time comparison of *CPMMX* and baselines on RealWorld-BSC-FOT dataset at block 25543755.

	Echidna	**ItyFuzz**	**Ours**
Precision	0.78	0.96	**1.00**
Recall	0.09	0.49	**0.93**
F1 Score	0.16	0.65	**0.97**
Vulnerable Time (min)	2290	1707	**150**
Benign Time (min)	2411	2422	**1867**
Overall Time (min)	4701	4129	**2017**
Vulnerable Timeout #	111.33	62	**6**
Benign Timeout #	119	119.67	**83.67**
Overall Timeout #	230.33	181.67	**89.67**

Table 10. Performance of *CPMMX*, *CPMMX-NoFee*, and *CPMMX-NoFeeNoRepeat* in detecting bugs in the DeFiHack-Labs dataset.

	# Bugs Detected	**Average Time (sec)**
CPMMX	**21**	51
CPMMX-NoFee	20	43
CPMMX-NoFeeNoRepeat	14	**31**

A.1 Evaluation on Fee-on-Transfer Tokens

To evaluate performance on challenging cases, we test *CPMMX*, ItyFuzz and Echidna on RealWorld-BSC-FOT, a modified version of the RealWorld-BSC in which all benign tokens are fee-on-transfer tokens. In the original RealWorld-BSC, only 22 out of 122 benign tokens have this property. Fee-on-transfer tokens deduct a portion of each transfer as a fee. In some cases, these fee mechanisms remove more tokens from DEXes than expected, causing **Invariant 1** violations. However, these tokens cannot be exploited to extract stablecoins from DEXes because they also implement safeguards. This makes distinguishing between benign fee-on-transfer tokens and CPMM composability bugs more difficult than differentiating non-fee tokens from CPMM composability bugs.

As shown in Table 9, although *CPMMX* continues to outperform the baselines, the gap in execution time has narrowed. *CPMMX* maintains a precision of 1.00, but its overall execution time increased from 597 minutes in Table 6 to 2017 minutes. This is because many benign fee-on-transfer tokens triggered **Invariant 1** violations. Echidna and ItyFuzz demonstrated similar performances.

A.2 Evaluation with DEXes Modified to Remove Fees

An alternative approach to detect CPMM composability bugs is executing template test cases with DEXes modified to remove fees. This method is commonly used in real-world bug finding, as it can eliminate the need for repetitions required to identify vulnerabilities. For instance, in the case of the ANCH bug (§3.1), *CPMMX* requires repetitions for the attacker to accumulate enough bonus to offset the DEX fee. If the DEX fee is removed, this bug can be detected without repetitions. To evaluate the effectiveness of this approach, we compare *CPMMX* with two variants using the DeFiHackLabs dataset: *CPMMX-NoFee*, which eliminates the DEX fee, and *CPMMX-NoFeeNoRepeat*, which removes both the DEX fee and the deep search phase for repetition.

As shown in Table 10, removing DEX fees resulted in lower bug detection rates. *CPMMX-NoFee* and *CPMMX-NoFeeNoRepeat* missed one more and seven more bugs compared to *CPMMX*. *CPMMX-NoFee* was unable to detect the vulnerability in GPT. After patching the DEX bytecode to remove fees, all test cases for GPT reverted. This likely results from GPT internally calling DEX functions and expecting fee-based behavior. *CPMMX-NoFeeNoRepeat* failed to detect CPMM composability bugs in GPT as well as in six other tokens: BFC, BRA, HEALTH, PLTD, Starlink, and TGBS. Even after removing DEX fees, these cases still required repetitions for profit. For instance, the PLTD token has both a fee and a bonus mechanism. Thus, the attacker must accumulate sufficient bonuses to offset PLTD's fees. On average, *CPMMX-NoFeeNoRepeat* is the fastest at detecting bugs, followed by *CPMMX-NoFee* then *CPMMX*. This suggests that removing DEX fees can be used as an optimization strategy to accelerate the bug detection process while maintaining reasonable accuracy.

References

[1] 2024. BscScan: The Binance Smart Chain Explorer. https://bscscan.com/ Accessed: 2024-10-24.

[2] 2024. Etherscan: The Ethereum Blockchain Explorer. https://etherscan.io/ Accessed: 2024-10-24.

[3] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2021. Uniswap Protocol Whitepaper. https://docs.uniswap.org/whitepaper.pdf Accessed: 2024-10-24.

[4] Ancilia Inc. 2022. Blockchain Security for Web3 Developers. https://x.com/AnciliaInc/status/1557846766682140672. Accessed: 2024-10-16.

[5] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. 2023. Clockwork finance: Automated analysis of economic security in smart contracts. *2023 IEEE Symposium on Security and Privacy (SP)*, 2499–2516. doi:10.1109/SP46215.2023.10179346

[6] BlockSec. 2023. BlockSec Twitter. https://twitter.com/BlockSecTeam/status/1624077078852210691.

[7] BlockSec. 2024. BlockSec. https://blocksec.com/.

[8] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 940–940. doi:10.1145/3597503.3639173

[9] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2024. Flashsyn: Flash loan attack synthesis via counter example driven approximation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639190

[10] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying invariant effectiveness for securing smart contracts. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1772–1795. doi:10.1145/3660786

[11] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021* (2021), 227–239. doi:10.1109/ASE51524.2021.9678888

[12] DeFiHackLabs. 2020. DeFiHackLabs. https://github.com/SunWeb3Sec/DeFiHackLabs. GitHub repository.

[13] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. doi:10.1109/wetseb.2019.00008

[14] Foundry. 2024. Foundry. https://github.com/foundry-rs/foundry. GitHub repository.

[15] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 945–956. doi:10.1109/ICSE48619.2023.00087

[16] Google. 2024. American Fuzzy Lop (AFL) - A security-oriented fuzzer. https://github.com/google/AFL Accessed: 2024-10-24.

[17] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 557–560. doi:10.1145/3395363.3404366

[18] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2023. Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1793–1810. https://www.usenix.org/conference/usenixsecurity23/presentation/gritti

[19] Sujin Han. 2025. Kaist-hacking/cpmmx: Artifact Release for ISSTA 2025. artifact-release, Zenodo. doi:10.5281/zenodo.15223116

[20] kaist-hacking. 2025. *CPMMX: Automatic Attack Synthesis for Constant Product Market Makers*. https://github.com/kaist-hacking/CPMMX Accessed: 2025-04-09.

[21] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1144–1156. doi:10.1145/3597926.3598124

[22] Shaun Paul Lee. 2023. Market Share of Decentralized Crypto Exchanges, by Trading Volume. https://www.coingecko.com/research/publications/decentralized-crypto-exchanges-market-share.

[23] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 752–764. doi:10.1145/3533767.3534222

[24] Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. 2024. SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 954–954. doi:10.1145/3597503.3639213

[25] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts.

In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. doi:10.1109/ASE.2019.00133

[26] Neptune Mutual. 2023. How Was BRA Token Exploited? https://medium.com/neptune-mutual/how-was-bra-token-exploited-24ff323249d.

[27] Mythril. 2017. Mythril. https://github.com/Consensys/mythril. GitHub repository.

[28] QuilAudits. 2022. ShadowFi $301K Burn function Exploit Analysis|QuilAudits. https://medium.com/quillhash/shadowfi-301k-burn-function-exploit-analysis-quillaudits-45a17ce04193.

[29] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333. doi:10.1145/3597926.3598059

[30] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639117

[31] SunWeb3Sec. 2023. BIGFI_exp.sol: DeFi Hack Labs Exploit Test Script. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2023-03/BIGFI_exp.sol Accessed: 2024-10-24.

[32] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21, 2 (2022), 515–529. doi:10.1109/TDSC.2022.3182373

[33] Mingxi Ye, Xingwei Lin, Yuhong Nan, Jiajing Wu, and Zibin Zheng. 2024. Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 794–805. doi:10.1145/3650212.3680321

[34] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. Ethploit: From Fuzzing to Efficient Exploit Generation Against Smart Contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 116–126. doi:10.1109/SANER48275.2020.9054822

[35] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. 2023. Your Exploit is Mine: Instantly Synthesizing Counterattack Smart Contract. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1757–1774. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhuo-exploit

Received 2024-10-31; accepted 2025-03-31