

Price-oracle manipulation: what a pre-deployment check can catch, and what it can't

MagiChain Security · October 2026

Type: a detection design and evaluation plan. No measured results yet.

Scope: EVM smart contracts that read a price and act on it (lending, vaults, AMMs, derivatives).

Price-oracle manipulation accounts for roughly \$300M in losses across the five incidents we review here. We sort it into four classes by where the price goes wrong, give a detection rule precise enough for a static analyser, a prover and a fuzzer to share, and set out how six independent validators divide the work. Three of the four classes are visible in contract code before deployment. The fourth is not, and a pre-deployment check should say so rather than pretend.

1 What we set out to answer

Many of the largest DeFi exploits are not bugs in the usual sense. The code does what it says; it just trusts a number it shouldn't. That number is a price read from a source the attacker can move, within one transaction or across a few blocks, and then fed into collateral valuation, a borrow limit or a mint. Flash loans make the capital for this free for the length of a transaction [1,2].

Our earliest design work used this attack as its running example. This note turns it into something we can build and measure, by answering three questions:

- Which oracle weaknesses can a check see in contract code before deployment, and which can't it?
- What exactly should the detection rule be, so that different engines agree on what they are looking for?
- How should six different validators split the work, and how do their results become one verdict?

This is the first note in our smart contract verification track. Our other research covers AI-agent payments.

2 Four ways a price goes wrong

Incident	When, chain	Reported loss	What the protocol trusted	Class
bZx (2nd attack)	Feb 2020, Ethereum	≈ \$0.6M	A DEX spot price, skewed with a flash loan in the same transaction [1]	A
Harvest Finance	Oct 2020, Ethereum	\$33.8M drained, \$2.5M returned	A Curve pool price used to value vault deposits and withdrawals	A
Cream Finance	Oct 2021, Ethereum	≈ \$130M	A vault's price per share, inflated by a donation and used as collateral value	B
Inverse Finance	Apr 2022, Ethereum	\$15.6M borrowed	A TWAP oracle with a window-size and sampling error	C
Mango Markets	Oct 2022, Solana	≈ \$116M	An oracle that correctly tracked a spot market the attacker had pushed up	D

Figures as reported in post-mortems and incident analyses [12–16]. Sources differ, and some figures are borrowed value rather than net loss.

A. Same-transaction spot price. The contract reads a price straight from pool reserves or balances (`getReserves`, `slot0`, a `balanceOf` ratio). A flash loan moves it, the protocol acts on it, and the loan restores it, all inside one transaction. *Visible in code: yes.*

B. Manipulable exchange rate. The price comes from a ratio the attacker can change by donating tokens or minting shares, such as `pricePerShare` or `totalAssets/totalSupply` computed from raw balances. *Visible in code: yes.*

C. Weak averaging. The contract does use a time-weighted average price, but the window is short or the sampling is wrong, so a few blocks of sustained pressure are enough. *Visible in code: partly.* The window is in the code; whether it is long enough depends on how deep the pool is.

D. Honest oracle, thin market. The oracle reports the market correctly. The market itself is cheap to move. *Visible in code: weakly.* Code can show that a collateral asset has no liquidity floor or position cap, but not how thin its market is. That is a runtime, economic question.

3 The detection rule

A contract is flagged when a value derived from a manipulable price source reaches a sensitive sink, and no adequate guard for that source runs on every path in between:

$$Flag(s, k) := Src(s) \wedge Sink(k) \wedge MayFlow(s, k) \wedge \neg \exists g : Dom(g, k) \wedge Covers(g, s)$$

Sources are reserve and balance reads, share-price reads, TWAP reads and external feeds. **Sinks** are collateral valuation, borrow limits, liquidation, mint and redeem amounts, and swap outputs. *MayFlow* over-approximates data flow, so no real flow is missed. *Dom(g, k)* requires the guard to dominate the sink: a guard on one branch is not a guard. *Covers(g, s)* requires the guard to be about this source, not some other price. A guard covers a source if it is one of:

<i>TWAP(s)</i> : value read from a cumulative-price accumulator over a window $w \geq w_{\min}$	(A, C)
<i>Dev(s)</i> : $ p_s - p_r \leq \delta \cdot p_r$, with reference r independent of s	(A, B)
<i>Multi(s)</i> : median of at least three mutually independent sources	(A, C)
<i>Acct(s)</i> : share price computed from internal accounting, not raw token balances	(B)
<i>Liq(s)</i> : reserves of the market behind $s \geq \ell$, with exposure caps on that asset	(C, D)

Three details matter in practice. First, independence: a deviation check against a reference the attacker can move in the same transaction checks nothing. Second, parameters: a one-block TWAP is a spot price under another name. The rule therefore reports w , δ and ℓ as findings instead of treating any guard as sufficient. Thresholds are a policy choice we make visible, not a truth we assert. Third, this rule is in the style of Securify’s compliance and violation patterns [3], and of tools that target price manipulation specifically [5,6]. What we add is the class split and the parameter reporting, which together say *how* a guard falls short, not just whether one exists.

4 Six validators, one verdict

No single analysis technique covers every class at an acceptable false-positive rate; a large benchmark of existing tools found that, combined, they detected fewer than half of the annotated vulnerabilities [9]. We therefore run six validators that fail in different ways:

Validator	What it contributes	Where it is blind	Classes
Classic signatures	Matches code cloned from exploited protocols, and known-bad deployers, in milliseconds	Anything new	A, B
Static analysis	Finds every source-to-sink flow and checks guard dominance [4,5]	Over-approximates; can’t judge whether δ or w is adequate	A, B, C
Formal verification	Proves the guard holds on every path, or returns a counterexample path	Needs a written property; costly on large code	A, B, C
Dynamic analysis	Replays a flash-loan or multi-block manipulation on a forked chain and measures the profit	Needs a harness; covers only the paths it runs	A, B, C, D (partly)
Deep learning model	Scores unseen variants, including bytecode with no source [10]	Opaque; only as good as its labels	A, B
LLM code analysis	Reads what a value is <i>for</i> (collateral or display) and explains the fix [7,8]	Not deterministic; can be confidently wrong	Triage, all

From findings to a verdict. Cheap engines run first: signatures, static analysis and the model’s score. The LLM then triages their output and explains it. The expensive engines confirm. A finding becomes a *violation* only when a confirming engine agrees, meaning a counterexample from the prover or a profitable replay from dynamic analysis. Otherwise it stays a *warning*, with the guard parameters attached. The aim is that one engine’s false positive can’t block a deployment, and one engine’s blind spot can’t let one through.

5 How we will measure it

- **Corpus.** Reproductions of real incidents, in the style of DeFiHackLabs [11], each paired with the protocol’s patched version. Pre- and post-fix pairs are the fairest false-positive test, since the code is nearly identical. Unrelated audited contracts serve as further controls.
- **Metrics, per class and per engine.** Detection rate on vulnerable versions; false-positive rate on patched versions; time per contract; and how often a warning reaches confirmation.

- **Reporting.** Rates across the corpus, never a single anecdote.

Hypotheses (*not yet verified*): **H1**: no single validator detects all of classes A–C at an acceptable false-positive rate, but the combination does. **H2**: vulnerable and patched versions separate cleanly for A and B, less so for C, where adequacy depends on parameters. **H3**: class D is not reliably detectable before deployment. The best a pre-deployment check can do is flag a missing liquidity floor or exposure cap.

6 Beyond this note

- **Agent payments.** An agent that acts on a DeFi price faces the same question as a lending contract: does the attacker control this number? The price-and-terms check in our agent-payment work and the rule above can share one verdict engine.
- **Patterns from post-mortems.** Every new incident becomes a rule and a pre/post-fix test pair, so the rule set grows with the threat landscape.
- **Deployer and wallet reputation.** A trust signal on who is deploying or being paid, alongside the code checks.
- **Runtime monitoring** for class D, where the risk lives in the market rather than the code.
- **Proof that the check ran.** Attach a validity proof to the verdict, so the chain can verify a contract was checked before it was deployed.

References

- [1] K. Qin, L. Zhou, B. Livshits, A. Gervais. Attacking the DeFi ecosystem with flash loans for fun and profit. *Financial Cryptography (FC)*, 2021.
- [2] L. Zhou et al. SoK: Decentralized Finance (DeFi) attacks. *IEEE S&P*, 2023.
- [3] P. Tsankov et al. Securify: Practical security analysis of smart contracts. *ACM CCS*, 2018.
- [4] J. Feist, G. Grieco, A. Groce. Slither: A static analysis framework for smart contracts. *WETSEB*, 2019.
- [5] Q. Kong et al. DeFiTainter: Detecting price manipulation vulnerabilities in DeFi protocols. *ISSTA*, 2023.
- [6] B. Wang et al. BlockEye: Hunting for DeFi attacks on blockchain. *ICSE (Demonstrations)*, 2021.
- [7] Y. Sun et al. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. *ICSE*, 2024.
- [8] Detecting various DeFi price manipulations with LLM reasoning. arXiv:2502.11521, 2025.
- [9] T. Durieux, J. F. Ferreira, R. Abreu, P. Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. *ICSE*, 2020.
- [10] Y. Zhuang et al. Smart contract vulnerability detection using graph neural networks. *IJCAI*, 2020.
- [11] DeFiHackLabs: reproductions of DeFi incidents with Foundry. <https://github.com/SunWeb3Sec/DeFiHackLabs>
- [12] bZx, February 2020: The Block, 17 Feb 2020; CryptoPotato, Feb 2020.
- [13] Harvest Finance, October 2020: The Block, 26 Oct 2020, citing Harvest’s post-mortem.
- [14] Cream Finance, October 2021: Immunefi, *Hack analysis: Cream Finance Oct 2021*.
- [15] Inverse Finance, April 2022: Inverse Finance, *INV price manipulation incident*.
- [16] Mango Markets, October 2022: CertiK, *Mango Markets incident analysis*.