



École Polytechnique Fédérale de Lausanne

Right Round:
LLM-Guided SMT Solving for
Arithmetic-Precision Bugs in DeFi

by Franco Sainas

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. sc. ETH Hubert Ritzdorf
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 21, 2026

Acknowledgments

I thank Prof. Mathias Payer for advising this thesis at EPFL and for the feedback that shaped its structure.

A special thanks goes to Hubert Ritzdorf for the supervision of this work at ChainSecurity. The idea for `/right-round` is his. He gave me the code and the questions from live reviews, read the drafts of this thesis, and always knew which ideas were worth chasing and which were not.

I also thank my colleagues at ChainSecurity, who tried the tool on their own work and helped me improve it.

Infine, ringrazio la mia famiglia, che mi sostiene in tutto.

Lausanne, August 21, 2026

Franco Sainas

Abstract

Most smart contracts in DeFi use integer arithmetic. Integer division can lead to precision loss. Divisions are common in DeFi to compute conversions, fees, and rates. Attackers trigger precision losses to violate safety properties in smart contracts. In 2025, three incidents caused by precision loss led to a cumulative theft of \$146.6 million. These bugs are hard to find for security experts and existing tools.

We present `/right-round`, given a codebase and a property in natural language, `/right-round` determines whether this property holds, providing helpful evidence. `/right-round` is designed to be easy to use and to support codebases with real-world complexity. `/right-round` is based on coding agents and leverages SMT solvers.

We evaluate `/right-round` on a benchmark of 288 test cases over 22 popular production repositories. The evaluation compares two different coding agents. Claude Code returns the expected verdict on 95.1% of the cases, in an average time of 19 minutes each. The main limitation is the combination of properties and codebases resulting in exceptionally complex nonlinear integer arithmetic. This arithmetic is undecidable, leading to `/right-round` not producing a verdict.

When used in real-world security reviews, `/right-round` uncovered real bugs, proving that it can provide a faster and deeper understanding of precision loss in smart contracts.

Contents

Acknowledgments	2
Abstract	3
1 Introduction	6
1.1 Right-Round	7
1.2 Contributions	9
2 Background	10
2.1 Ethereum and Decentralized Finance	10
2.2 Common DeFi Protocols	11
2.3 Integer Arithmetic and Rounding	11
2.4 Precision-sensitive Accounting in DeFi	12
2.5 SMT Solving and Z3	12
2.6 Coding Agents and Skills	13
3 Related Work	14
3.1 Assurance Techniques for Smart Contracts	14
3.2 Why This Bug Class Escapes Existing Techniques	15
3.2.1 Fuzzing	15
3.2.2 Formal Verification	15
3.2.3 Static Analysis	16
3.2.4 Rounding-specific Analysis	16
3.2.5 The Remaining Gap	16
4 Design	17
4.1 Approach	18
4.2 Goals	19
4.3 Technical Contributions	19
4.3.1 C1: the /right-round skill	20
4.3.2 C2: the faithfulness method	21
4.3.3 Supporting Components	21

4.4	Architecture and Workflow	22
5	Implementation	25
5.1	The SMT Encoding	25
5.2	The Faithfulness Check	28
5.3	The Proof of Concept	29
5.4	The Artifacts	29
5.5	The Plugin	30
6	Evaluation	33
6.1	The Benchmark	33
6.2	Experimental Setup	34
6.3	Measuring Evidence Quality	35
6.4	Effectiveness (RQ1)	36
6.5	Cost (RQ2)	39
6.6	Limitations (RQ3)	41
6.6.1	Solver Intractability	41
6.6.2	Bounded Loop Unrolling	42
6.6.3	Scope and Bounds	42
6.6.4	From Natural Language to Predicate	42
6.6.5	Limitations of the Faithfulness Method	42
6.6.6	Overall	43
7	Real-World Case Studies	44
7.1	Audit of a Morpho Integrator	44
7.2	Deeper Understanding of Rebasing Tokens	46
7.3	Performance in Case Studies	47
8	Conclusion	48
8.1	Future Work	49
8.1.1	The Solver as Confirmer	49
8.1.2	Other Source Languages	49
	Bibliography	50
A	Benchmark Fixtures	54

Chapter 1

Introduction

Decentralized finance (DeFi) replaces the intermediaries of a financial system with programs that run on a public blockchain. These programs are called *smart contracts*. Their code and state are public, which shifts trust from brokers, exchange operators, and custodians to code that can be inspected [38]. At the time of writing, DeFi protocols across all chains hold \$79 billion of value locked [19]. The majority of DeFi runs on chains compatible with the *Ethereum Virtual Machine* [39], or EVM.

The absence of trusted parties makes the code mission critical, as no one can reverse a mistake. In 2025, the total losses traceable to smart contract vulnerabilities were \$512 million [26]. Transactions are irreversible, making lost funds very hard to recover.

Close to a third of the total losses came from a single bug class: numerical precision loss. The EVM executes smart contract bytecode and offers only integer operations. Contracts divide frequently, whenever they compute a share, a rate, or a fee. When an integer is divided the remainder is discarded and the exact result is *rounded* down. The developer can choose to round up. A division has an error of at most one unit. This becomes dangerous when the error is repeated many times, when it is subsequently multiplied, or when the contract is in a state where one unit carries a large value.

In 2025, a rounding error in the balance scaling of Balancer V2 was compounded across 65 swaps inside a single transaction, leading to the loss of \$128.6 million [15]. An inflated lending accumulator combined with withdrawal rounding drained \$9.6 million from zkLend [33]. A sequence of 44 tiny withdrawals against Bunni accumulated an error leading to mispriced trades and a loss of \$8.4 million [10]. Those three incidents total \$146.6 million.

These bugs survive the review process the industry already applies, called an *audit*. Precision loss is easy to see and hard to judge: an analyst reading the source sees that a division rounds and in which direction, but whether the error is dangerous depends on how the result is used. For

Bunni’s withdrawal operation the developers had chosen what they considered “the safe rounding direction” [10]. It was considered safe with a certain safety property in mind. However, what they did not consider was that an accumulated precision loss violated a different safety property when the contract is in a degenerate state. This highlights how hard it is for experts to reason about bugs of this class.

We analyzed how easy it is to find such bugs with existing tools under audit time pressure, using static, symbolic, and dynamic analysis. Existing tools fail in five recurring ways, which we refer to throughout this thesis as failure modes:

Unfaithfulness The tool tests something semantically different from the original code.

Friction The tool requires significant time for onboarding, usage, or evidence interpretation when used on a complex codebase.

State explosion The tool reasons about too many states at the same time, failing.

Unreachability The tool does not test the harmful states.

Inexpressivity The tool cannot state the property of interest.

In line with our claim that existing tooling is insufficient, a study of 127 high-impact DeFi exploits found that the state-of-the-art automated tools would have prevented only 8% of them, covering \$149 million of \$2.3 billion in losses [14], which leaves expensive expert review as the mechanism that has to catch the rest.

1.1 Right-Round

To help experts reason about this bug class, we present `/right-round`. The analyst supplies a codebase and a property in natural language, such as “depositing assets and then redeeming the resulting shares never returns more than was deposited”, and they receive back a report that explains whether the property holds. If the property is violated, `/right-round` builds a proof of concept (PoC) that triggers the violation on the real codebase; otherwise, it provides a proof of why the property holds. This process is designed to be effortless, saving the analyst as much time as possible, while giving stronger guarantees about the absence of bugs. Therefore, we believe that `/right-round` can improve the understanding of this bug class while reducing the time needed to analyze it, filling an existing gap in the industry.

`/right-round` is implemented as a skill for coding agents. As shown in Figure 1.1, the analyst supplies a property and gives access to a codebase. The coding agent reads the code and locates the functions whose arithmetic determines whether the property of interest holds. Then it encodes the

arithmetic operations as constraints, and performs a faithfulness check. It asks an SMT solver to check the satisfiability of the constraints. Finally, it generates the report. The report is designed to give the analyst a deeper understanding of the investigation, in a concise form.

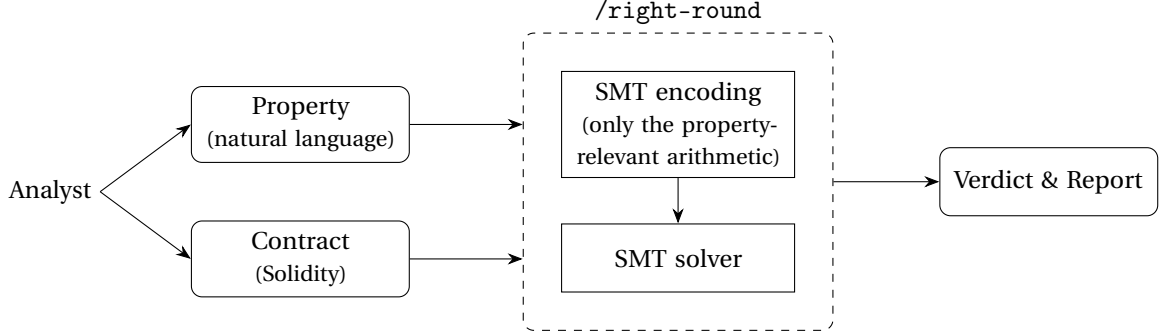


Figure 1.1: Overview of /right-round.

/right-round addresses the five failure modes, as summarized in Table 1.1.

Failure mode	How /right-round addresses it
Unfaithfulness	The original contract and the LLM-built encoding are checked to agree using differential fuzzing.
Friction	The analyst states a property in natural language, which results in fast usage and little onboarding. The report allows efficient evidence interpretation.
State explosion	The SMT encoding includes only the relevant arithmetic, reducing the number of constraints.
Unreachability	The state of the original contract is encoded as free variables, exploring every configuration.
Inexpressivity	Properties about any internal or intermediate state can be stated.

Table 1.1: How /right-round addresses each of the five failure modes of existing tools.

To evaluate /right-round, we ask three questions:

RQ1 (Effectiveness) On a benchmark with real-world codebases and properties, how often does /right-round return the correct verdict with good evidence?

RQ2 (Cost) What does running /right-round cost in time, tokens, and money?

RQ3 (Limitations) What are the limitations of this approach?

By performing audits of a variety of projects in the industry for multiple months, we obtained the necessary domain knowledge to form the ground truth for a benchmark. We evaluate /right-round

on a benchmark of 288 test cases over 22 popular production codebases, among them OpenZeppelin, Solady, Uniswap, Aave, Morpho, and Compound, spanning vaults, rebasing tokens, lending markets, and automated market makers. The effectiveness of `/right-round` heavily depends on the coding agent used. Claude Code, the best coding agent, returns the expected verdict on 95.1% of the cases. Furthermore, on 80.6% of the cases it also produces flawless evidence. For Claude the average cost is \$9.48 per case. We found the major limitation to be exceptionally complex nonlinear integer arithmetic. This limitation is inherited from the SMT solver and relevant for multiple DeFi projects.

To evaluate its real-world usefulness, `/right-round` was used at ChainSecurity, a smart contract auditing company. `/right-round` produced findings during audits that were later confirmed by experts. Furthermore, while being used to get a deeper understanding of the rounding behavior of rebasing tokens for a blog article [13], it produced two findings on live contracts holding over \$500 million. Along with the findings, `/right-round` delivered helpful evidence in the form of PoCs. These findings and the PoCs were responsibly disclosed to the affected teams and acknowledged by them.

1.2 Contributions

This thesis makes the following contributions:

- `/right-round`, an LLM-guided SMT workflow that checks safety properties defined in natural language (chapter 4, chapter 5).
- A faithfulness check that the LLM-generated SMT encoding matches the original code (subsection 4.3.2).
- A benchmark of 288 test cases over 22 real-world production codebases, with properties and expected verdicts (section 6.1).
- An empirical evaluation of the implementation, spanning effectiveness, cost, and limitations across two coding agents (chapter 6).
- A per case evaluation procedure of the evidence quality using an LLM-judge (section 6.3).

The rest of the thesis covers the background on smart contracts, integer arithmetic, SMT solving, and coding agents (chapter 2), then the assurance techniques in use today and how they fail on this bug class (chapter 3). The chapters that follow present the design of `/right-round` and its technical contributions (chapter 4), its implementation (chapter 5), the answers to RQ1, RQ2, and RQ3 (chapter 6), the real-world case studies (chapter 7), and the conclusions with future work (chapter 8).

Chapter 2

Background

This chapter presents the concepts needed to understand this bug class, and surveys the available techniques to catch it.

2.1 Ethereum and Decentralized Finance

A *blockchain* is a replicated state machine with no privileged operator: its nodes agree on an ordered log of updates and each node applies all of them, so every node holds the same state. Ethereum is a public blockchain that runs arbitrary programs, called *smart contracts*. The state is a set of accounts, each holding a balance of ether, Ethereum's native currency. A contract account additionally holds code and storage. A common use of smart contracts is to track ownership of other assets: an ERC-20 *token* is a contract that keeps a mapping of balances in its storage and defines how they can be transferred [37].

Contract code and state are readable by anyone, and the state changes only through *transactions*, which can call contract functions. Anyone can send a transaction calling any external function with any input, and calls to several contracts can be composed within a single transaction that either commits or fails as a whole. Once executed, a transaction is irreversible. Contracts are also immutable: unless one is built to be upgradeable, patching a bug is not possible. Reviewing and assuring a contract before deployment is therefore critical.

Contracts are written in a high-level language such as Solidity and compiled to bytecode that runs on the *Ethereum Virtual Machine* (EVM) [39]. The EVM is a stack machine over 256-bit words, and it offers integer operations only.

Decentralized finance (DeFi) builds financial services from these contracts [38]. Each service, or *protocol*, is a set of contracts that holds the assets and enforces the rules, so no bank, broker,

or exchange operator stands in between. Protocols often retain privileged roles, but their powers are visible in the code and confined to the operations it defines. Typical services are vaults that invest deposits and issue shares, lending markets that lend against collateral and track debt, and automated market makers that price a swap from the pool's holdings of two tokens. These services compose with each other: protocols are routinely built on top of one another.

2.2 Common DeFi Protocols

This section introduces four common DeFi use cases.

Vaults issue shares against a pool of one asset (section 2.4). The ERC-4626 standard defines their interface [35].

Rebasing tokens pay interest by increasing the balances of their holders. Their contracts store each holder's shares and derive the balance on demand from an index that grows over time [28]. Every balance query and every transfer converts between shares and balance, and every conversion divides.

Lending markets let users supply an asset to earn interest, and borrow it against collateral. A borrow is allowed while the position stays healthy, that is, while the debt stays under the fraction of the collateral value set by a loan-to-value ratio. A position that fails the check can be liquidated.

Automated market makers, or AMMs, allow users to swap a token for another one. The liquidity used to serve the trade is provided by other users in exchange for trading fees. A simple pricing rule keeps the product of the two holdings constant [2]. Concentrated liquidity lets providers place liquidity in a price range delimited by *ticks* [3]. AMMs based on stableswap pools price a swap from an invariant that has no closed form and is solved iteratively [20].

2.3 Integer Arithmetic and Rounding

DeFi contracts constantly compute a conversion of one quantity into another. A deposit becomes vault shares. A debt is scaled by an interest index. Collateral is valued at an oracle price. A fee is a fraction of a trade. With only integers available, each of these is a multiplication followed by a division.

These conversions share the same shape: $a \cdot b / c$. The divisor is a scale constant when the rate is stored as a *fixed-point* value, an integer scaled by a constant S , commonly 10^{18} , and it is a quantity the contract tracks, such as a vault's total assets, when the result is a share of a pool. The operation is common enough to be a primitive in popular math libraries, called `mulDiv` in OpenZeppelin, Solady,

and PRBMath [9, 32, 36], and a protocol’s accounting is a chain of them.

The division truncates, so the result is not exact. Rounding down puts it below the exact value, rounding up puts it above, in both cases by less than one unit. Rounding down is what the machine already does, and rounding up has to be written out explicitly, which over integers is

$$\left\lceil \frac{ab}{c} \right\rceil = \left\lfloor \frac{ab + c - 1}{c} \right\rfloor.$$

The libraries expose both directions under paired names, for instance `mulDivDown` and `mulDivUp`, and the developer picks one at every call site.

2.4 Precision-sensitive Accounting in DeFi

Vaults convert between shares and assets on every deposit and withdrawal. Standard security practice is to round each division in the vault’s favor: converting deposited assets to shares rounds the shares down, and redeeming shares for assets rounds the assets down. Therefore, a deposit followed by a redemption never returns more than was put in.

This rounding enables the *first-depositor*, or *inflation*, attack. Consider an empty vault. The attacker deposits one unit and receives one share, so the vault holds one share against one asset. They then transfer 10^6 units of the asset directly into the vault, raising its balance to $10^6 + 1$ while the share supply stays at one. A victim who now deposits 10^6 units receives

$$\left\lfloor \frac{\text{assets} \cdot \text{totalShares}}{\text{totalAssets}} \right\rfloor = \left\lfloor \frac{10^6 \cdot 1}{10^6 + 1} \right\rfloor = 0$$

shares, yet their deposit has joined the pool, and the attacker redeems the single share for the vault’s entire balance [31]. One possible defense is *virtual shares*: the vault computes conversions as if it held a fixed offset of extra shares and assets that nobody owns, so a donation is partly absorbed by them and inflating the price enough to harm a depositor costs the attacker more than they could gain.

2.5 SMT Solving and Z3

An *SMT* (satisfiability modulo theories) solver decides whether a set of constraints over theories such as integer arithmetic has a satisfying assignment [7]; Z3 is one such solver [30]. To check an arithmetic property, one encodes the computation it concerns as constraints over integer variables and asks the solver whether any assignment of those variables violates the property. If the solver finds one, that assignment is a concrete counterexample (SAT); if it shows that none exists, the

property holds for every input the encoding allows (UNSAT).

A solver can also return UNKNOWN: on a timeout, or because the constraints fall in a theory with no complete decision procedure. Nonlinear integer arithmetic (multiplying unknowns, or integer division, which is encoded through multiplication) is one such theory, undecidable in general [29]; on it a solver falls back on incomplete heuristics and may return UNKNOWN rather than SAT/UNSAT.

2.6 Coding Agents and Skills

A *coding agent* is a large language model (LLM) driving a tool-use loop: given a task, the model issues an action, such as reading or writing a file or running a shell command, observes the result, and continues until it judges the task complete [40]. Agentic coding tools such as Claude Code expose this loop over a working directory, so the model can read a codebase, edit it, and run its tests.

A *skill* packages a repeatable procedure for such an agent. It is a bundle whose instruction file, loaded when the skill is invoked, states the steps and conventions to follow, alongside resources such as scripts, templates, and examples that the agent reads on demand as the steps require them [5]. A skill is invoked by name. `/right-round` is built as such a skill (see chapter 4).

Chapter 3

Related Work

In this chapter we discuss how existing tools can be used to find precision bugs. For each of the existing tools we point out which of the failure modes apply to them.

3.1 Assurance Techniques for Smart Contracts

Several automated techniques aim to catch precision bugs before deployment. This section surveys the main ones.

Fuzzing runs a contract on generated inputs and checks that a supplied property holds. Foundry, a popular Solidity development framework, provides a *stateless* fuzzer: it runs a test on randomly generated inputs, resetting state between runs. Echidna and Medusa are *stateful* fuzzers: they generate sequences of calls so that state accumulates across the sequence. The analyst writes a setup, handler functions exposing the target’s entry points (often restricting inputs to steer the search), and the invariants, which the fuzzer checks after each call [17, 22, 24].

Formal verification aims to prove a property for all inputs, though some tools guarantee this only up to a bound. Certora, a commercial prover, takes a separate specification in the Certora Verification Language (CVL) and proves the contract satisfies it. Halmos and Kontrol instead express properties as assertions inside test functions, run on symbolic inputs, exploring every path the bounds permit, up to bounded loop unrolling and call depth [1, 27, 34]. SolCMC (the Solidity compiler’s SMTChecker) checks the assertions written in the code together with the compiler’s built-in safety properties such as overflow and division by zero [4]. Certora and SolCMC have been compared on open Solidity benchmarks [8].

Some tools run without the analyst writing a property: Slither, a static analyzer, flags known vulnerability patterns through dataflow and taint tracking, and Mythril does so through symbolic

execution and SMT solving [16, 21].

A few tools address numerical precision directly. The roundme tool (Trail of Bits) takes a formula and a desired rounding direction for the result, and it reports which direction each operation must round to preserve that target. RoundAbout, a Certora static-analysis prototype (WALA-based), infers the rounding direction across variables, expressions, functions, and calls of a Solidity project and emits a per-public-function report [12, 18].

3.2 Why This Bug Class Escapes Existing Techniques

Beyond manual review, each automated technique misses arithmetic-precision bugs in a characteristic way.

3.2.1 Fuzzing

Fuzzers suffer from *unreachability*. Stateless fuzzers reset the state between runs, so they only test what the setup builds, never a call that depends on an earlier one or on time passing. Stateful fuzzers such as Echidna and Medusa accumulate state across a sequence of calls, and are coverage-guided: a newly reached branch rewards the search and steers it. Precision bugs offer no such signal. Reaching it often takes a long sequence of precise calls that accumulate the error, as in the 44 withdrawals against Bunni [10]. Without a gradient, more runs raise the hit probability only marginally.

3.2.2 Formal Verification

Verifiers such as Certora, SolCMC, and Mythril aim to reason over all possible inputs and states. They suffer from *friction*, *unfaithfulness*, *state explosion*, and *inexpressivity*.

Certora suffers mainly from friction and unfaithfulness. The prover works on specifications written in CVL. Learning this language requires significant onboarding time and the successful verification of a property largely depends on the familiarity the analyst has with the tool. Getting big codebases working with the tool is substantial work of dependency remappings, and the most complex contracts may not compile in the required form at all. The prover documents its own approximations, and warns they may be underapproximations that let bugs go unnoticed [11].

SolCMC suffers from state explosion and inexpressivity. Properties are written as `assert` statements instrumented into the contract [4]. Defining properties is often complex and error prone. Local variables are accessible only inside the functions they are defined in. Ghost code that tracks their value is needed to access them from outside or to track their intermediate states. This is usually needed to define a property over multiple functions. On a benchmark of 323 verification tasks

over small purpose-built contracts, 153 were not expressible in SolCMC, against 60 in Certora, and SolCMC timed out or exhausted memory on 22 more [8]. Those contracts are a fraction of the size of production code.

Mythril does not need a specification. Instead it checks a fixed catalog of vulnerability conditions. None of them covers precision: its arithmetic module looks only for overflows and underflows. A property can be supplied as an assertion in the code, so it suffers from the same expressivity limitations as SolCMC.

3.2.3 Static Analysis

Static analysis tools such as Slither require little effort to use. They need no specification, no harness, and none of the tool expertise fuzzing and formal verification require. A precision bug, however, is not a fixed pattern. Precision loss occurs in almost every division and cannot be avoided. Determining if a given instance is harmful depends on whether it violates an important invariant and the loss is large enough to be significant for the protocol. These tools have no way to state that condition, so they suffer from inexpressivity.

3.2.4 Rounding-specific Analysis

The roundme tool takes a formula and reports which way each operation must round to achieve a desired rounding direction. RoundAbout infers the rounding directions of arithmetic operations and propagates them across the project variables. However, knowing which way a value loses precision does not settle whether that rounding is harmful. That judgment is left to the analyst. RoundAbout also loses precision of its own: where a value rounds one way on one branch and another way on another, it merges the two and reports the direction as indeterminate. Its propagation follows only the calls the code itself makes, whereas the composition of interest may be one the source never writes, such as feeding the result of one external function into another, as an integrator or attacker would. These tools also suffer from inexpressivity.

3.2.5 The Remaining Gap

No existing approach offers what these bugs demand at once. Detecting one means reaching the exact state where a rounding turns harmful, reasoning about the arithmetic precisely, checking a property specific to the protocol, and staying faithful to the original code, all at a cost a code review can absorb. Fuzzing, verification, and static analysis each satisfy some of these requirements but not all. `/right-round` closes the gap by narrowing the problem: it encodes only the arithmetic the property depends on, and leaves the rest of the codebase out (chapter 4).

Chapter 4

Design

The difficulty of detecting arithmetic-precision bugs in real DeFi code traces to five failure modes of existing tools: *unfaithfulness*, *friction*, *state explosion*, *unreachability*, and *inexpressivity* (chapter 1, section 3.2). This chapter presents `/right-round`, a skill for coding agents that addresses all five.

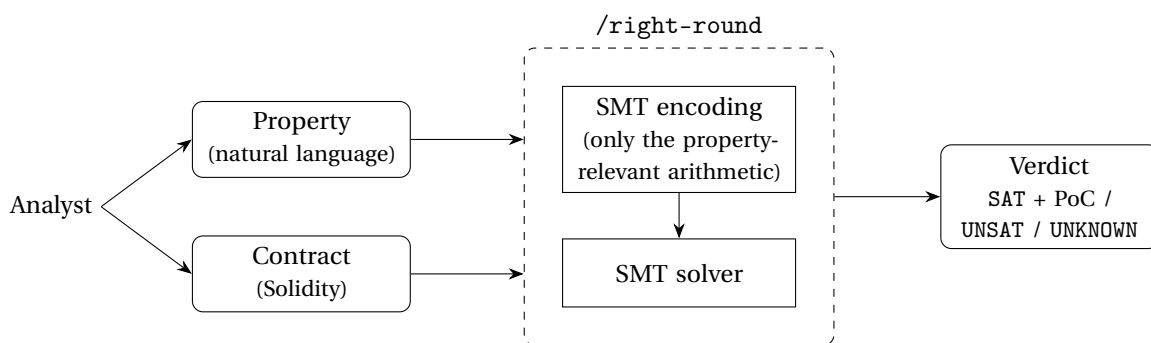


Figure 4.1: Overview of the approach for `/right-round`. Figure 4.2 details the pipeline inside the dashed box.

Figure 4.1 gives an overview of the workflow. The analyst supplies a property stated in natural language, and gives the coding agent access to the contracts it is supposed to hold for. From these, `/right-round` builds an *SMT encoding* that expresses both the arithmetic operations that determine whether the property holds and the property to be checked as constraints over free variables. The solver then searches for an assignment of those variables that satisfies the constraints; such an assignment is a counterexample to the property (SAT). Its absence means the property holds within the scope of the encoding (UNSAT). On a violation, the workflow also produces a runnable proof of concept (PoC). The rest of this chapter expands this outline into the approach (section 4.1), the goals (section 4.2), the technical contributions (section 4.3), and the full pipeline (section 4.4, Figure 4.2).

4.1 Approach

/right-round targets *safety properties that a precision bug can violate*. A *safety property* states that some bad outcome never occurs, so any violation is exhibited by a single finite execution. The properties of interest hold under exact arithmetic but can fail under any finite-precision arithmetic that must round, here the EVM's integer division, which discards the remainder. For example, converting assets to shares (a vault's internal accounting units) and back should be lossless:

```
shares = assets * totalShares / totalAssets; // rounds down
out    = shares * totalAssets / totalShares; // rounds down
```

Under exact arithmetic out equals assets. Because each division discards its remainder, out can be strictly smaller, violating the property.

Given such a property and the codebase, the workflow locates the relevant code. It collects the functions that implement the arithmetic determining whether the property holds, from the codebase and its dependencies. It produces three artifacts:

1. an *SMT encoding legible enough to inspect by hand*, for the analyst who wants to audit the reasoning directly;
2. a *verdict* (SAT, UNSAT, or UNKNOWN) and a report that records the property checked and justifies the verdict, explaining why a counterexample arises or why the property holds, so that the analyst can interpret the result efficiently; and
3. on SAT, a *runnable PoC* that triggers the violation on the real codebase.

Because the property is supplied in natural language, its translation into a formal predicate is an unverified step, and a verdict is only meaningful for the predicate actually checked. The report therefore records that exact predicate, so the analyst validates it by inspecting the report rather than the encoding. Direct inspection of the encoding is supported but optional. Natural-language input keeps the entry cost low, and the properties of interest are typically short. Three representative properties:

- “Depositing assets and then redeeming the resulting shares never returns more than was deposited.”
- “`borrow(maxBorrow(ltv))` always succeeds.”
- “After a full balance transfer, the sender’s aToken balance is exactly zero.”

Each is translated into a formal predicate before solving. The first, for instance, becomes $\text{redeem}(\text{deposit}(a)) \leq a$ for every deposited amount a , and the solver searches for an assignment that violates it.

The workflow takes on the mechanical SMT encoding, leaving to the analyst the one judgment a natural-language specification cannot resolve: whether the predicate actually checked corresponds to the intended property.

4.2 Goals

The design goal is a *grounded* verdict for such a property at *low friction*, together with a runnable PoC when the property is violated. *Grounded* rules out the plausible-but-wrong answers that ungrounded LLM-for-audit tools produce. *Low friction* rules out the hours of manual SMT encoding or harness construction that the alternatives demand. The goal decomposes into five sub-goals.

Construction sub-goals.

G1 (Workflow) Integrate code-reading, SMT encoding, solving, and PoC generation into a single standardized pipeline.

G2 (Faithfulness) Establish that the encoding computes what the contract computes, by testing it against real execution.

Evaluation sub-goals.

G3 (Effectiveness) Evaluate how often the workflow returns the correct verdict with good evidence, on real-world codebases (RQ1).

G4 (Cost) Quantify the cost to reach a verdict in time, tokens, and money (RQ2).

G5 (Limitations) Identify the limitations of the approach (RQ3).

The rest of this chapter is about the construction sub-goals (G1, G2); the evaluation sub-goals (G3–G5) are taken up in chapter 6.

4.3 Technical Contributions

The design has two main contributions, supported by a layer of machinery. The first is the *right-round* workflow; the second is the faithfulness method that makes its verdicts trustworthy. The supporting machinery makes the pipeline practical.

4.3.1 C1: the /right-round skill

Encoding a safety property for an SMT solver by hand is expert work that takes hours. /right-round reduces it to a single pipeline that reaches a verdict in minutes.

A coding agent can be prompted to perform the same task, but that takes sustained manual steering. The verdict depends on two artifacts the agent writes: the encoding and the faithfulness check that tests it. The analyst must understand and validate unfamiliar code each time if both are structured differently on every run. Beyond that, the agent makes small recurring mistakes that it lacks the tool-specific knowledge to avoid. For instance, we observed it encoding the arithmetic over bitvectors instead of integers, which leads to frequent timeouts, or checking the encoding against a partial reimplementation of the contract rather than the contract itself, which invalidates the check. Each of these errors must be caught and corrected on every run, and together they make reaching a trustworthy verdict slow and expensive, in analyst time and in tokens.

/right-round addresses both the effort and uncertainty of manual steering. The skill sets the steps and the encoding conventions, and supplies the tool-specific knowledge the agent lacks. It builds the encoding from an audited library of arithmetic primitives, instead of leaving the agent to reimplement them each time. The skill also guards against the pitfalls of assembling the artifacts, so the agent does not rediscover and re-correct the same errors on every run. Because the soundness-critical parts of the procedure are fixed and vetted, the analyst does not re-audit them each time. The report records the property checked and justifies the verdict, so the analyst validates the property and the verdict rather than the workflow behind them. The design addresses the four failure modes of existing tools in turn:

Unfaithfulness The faithfulness check verifies the encoding against the real compiled bytecode.

Friction Stating the properties in natural language does not require learning how to use a new tool or programming language. The report allows fast interpretation of the results.

State explosion The encoding is a small set of constraints, so the solver operates over a much smaller problem.

Unreachability The state variables of the contract are encoded as free variables, and the solver searches the constrained space symbolically.

Inexpressivity The state and local variables are defined as free variables, allowing properties to be stated about any internal or intermediate state.

4.3.2 C2: the faithfulness method

An LLM-built encoding can silently diverge from the source: a wrong rounding direction, a missing library function, an incorrect parameter. Any of these yields an encoding that is internally coherent and solvable yet no longer describes the contract. The faithfulness method grounds the encoding in the real code by testing it differentially against execution, through two complementary checks.

Fuzzing the encoding against the real compiled bytecode catches divergence on reachable inputs: if the two disagree on any sampled input, the encoding is wrong. Mutation testing addresses the risk that the check passes only because it is too insensitive to detect a discrepancy. Each rounding direction in the encoding is inverted and the faithfulness check is required to reject the resulting mutant, since a check that accepts an encoding known to be faulty provides no evidence of faithfulness. Together the two checks establish that the encoding agrees with the bytecode on the sampled inputs.

4.3.3 Supporting Components

Five supporting components make the pipeline practical:

Fixed-point primitive library A shared, audited library (`fixedpoint.py`) encapsulates the fixed-point operations the target contracts use, such as `mulDivDown` and `mulDivUp`. Encodings are composed from these validated components rather than reimplementing each operation, removing a recurring source of encoding error and inter-run variance.

Lemma-based UNSAT proofs When the solver times out on a property, the workflow decomposes it into smaller lemmas, proves each independently, and re-solves the property with the proved lemmas asserted as assumptions (Figure 4.4), decomposing further if the re-solve still times out. It gives up with UNKNOWN when no useful decomposition remains or the problem is fundamentally intractable for an SMT solver, such as nonlinear integer arithmetic (NIA). In practice this discharges the property (UNSAT); a SAT re-solve is possible but we have not observed one.

Free-variable bound management The companion skill `/bounds` lets the analyst inspect and adjust the bounds on the encoding's free variables and re-solve. The analyst may know realistic bounds that the encoding does not capture and can impose them. Because an UNSAT verdict holds only within the bounds set, making them explicit states exactly the domain over which the property is proved to hold. Tightening the bounds after a SAT tests whether the counterexample survives in a realistic range or was an artifact of an over-wide domain.

PoC construction When the verdict is SAT, the solver returns a counterexample as concrete input values. `/right-round` builds a runnable test that supplies these values to the real contract

and checks that the violation occurs, turning the abstract counterexample into an executable proof of concept.

PoC anchoring A counterexample rarely corresponds to a reachable on-chain state. The companion skill `/anchor` constrains the encoding with live on-chain values and re-solves for inputs consistent with the deployed state, so the constructed PoC executes against the live contract.

4.4 Architecture and Workflow

Figure 4.2 shows how the components are connected. Reading and extraction narrow the codebase to the relevant arithmetic; the workflow builds the encoding from that arithmetic using the primitive library; the faithfulness check gates the encoding before it is solved. If the encoding and the bytecode disagree, the workflow rebuilds the encoding; if a planted mutation survives, it rebuilds the test. Only an encoding that has passed the gate reaches the solver, which returns one of three verdicts.

On UNSAT, the property holds within the scope of the encoding. On SAT, the workflow constructs a runnable PoC from the counterexample. On UNKNOWN, the workflow could not decide the property. When the solver times out, it first attempts a lemma-based proof (Figure 4.4), and returns UNKNOWN only once that attempt is exhausted: when no useful decomposition of the property remains, or the property is fundamentally intractable for an SMT solver, such as nonlinear integer arithmetic (NIA).

The companion skills can be applied afterward. Each refines the encoding and re-solves it (Figure 4.3). `/anchor` constrains the free variables with live on-chain values, so a SAT yields a PoC that runs against the deployed contract and a previously UNKNOWN result may become decidable. `/bounds` adjusts the bounds on the free variables, for instance tightening them to test whether a SAT counterexample survives in a realistic range.

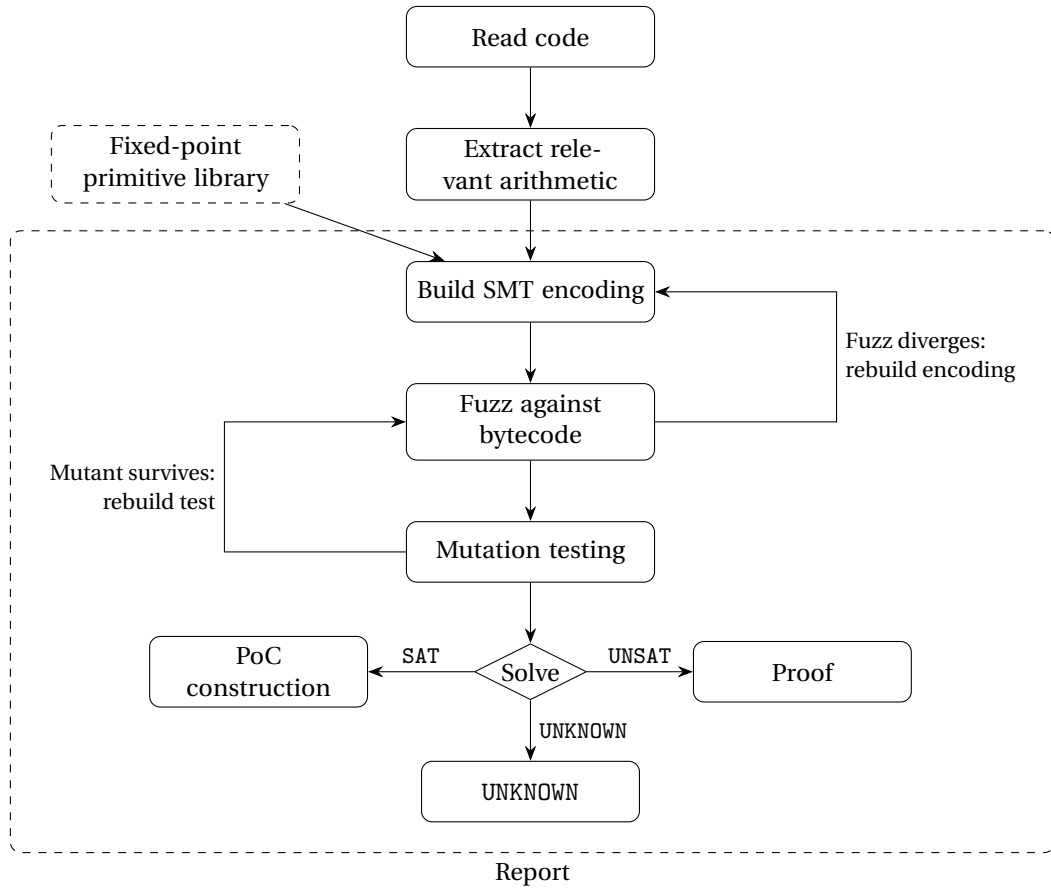


Figure 4.2: The /right-round pipeline.

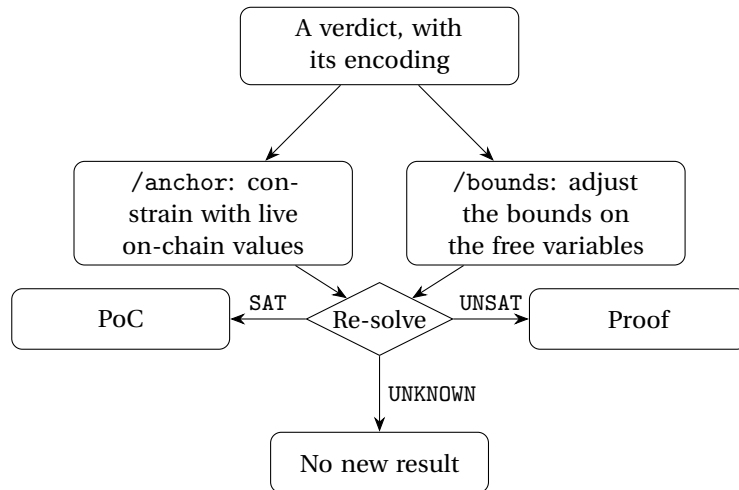


Figure 4.3: The /anchor and /bounds companion skills. Either refines the encoding, /anchor by constraining it with live on-chain values and /bounds by adjusting the bounds on its free variables, and the refined encoding is re-solved.

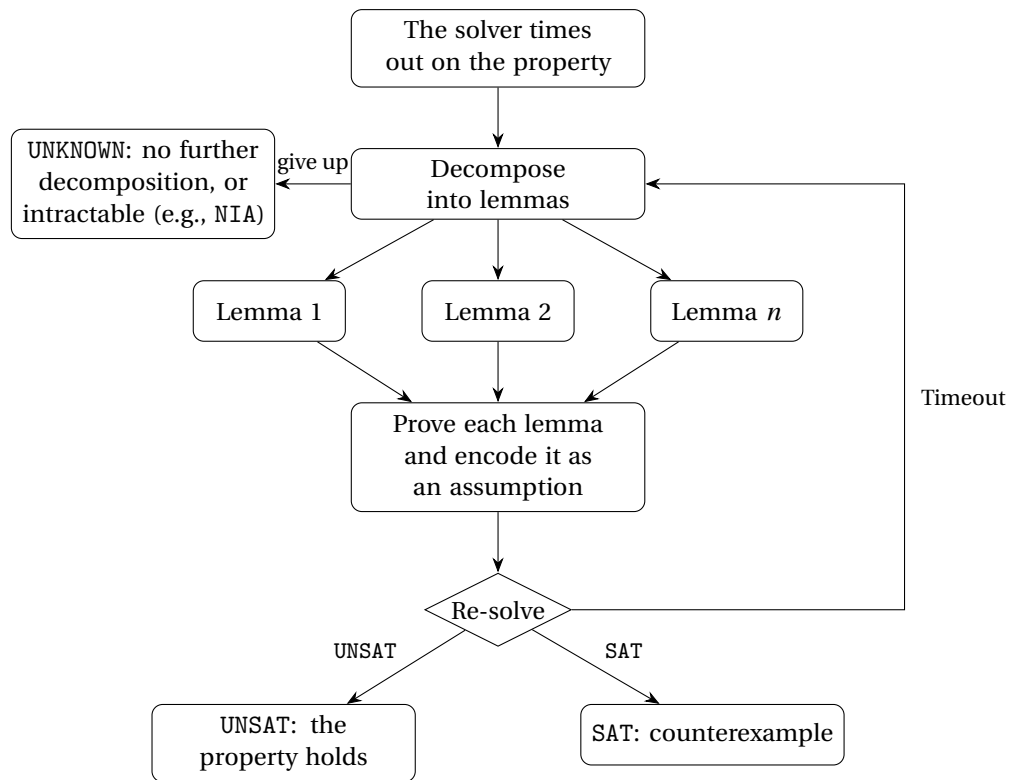


Figure 4.4: Lemma-based UNSAT proof.

Chapter 5

Implementation

`/right-round` is implemented as a skill for coding agents. This chapter covers how an encoding is built and the primitive library it draws on (section 5.1), the faithfulness harness (section 5.2), how a counterexample becomes a runnable proof of concept (section 5.3), the artifacts each run produces (section 5.4), and how the skill bundles and runs these steps (section 5.5).

5.1 The SMT Encoding

First, the agent gathers the arithmetic the property depends on. It reads the contract, and copies the functions which are relevant to the property being verified, together with the functions and constants they use from the rest of the codebase and its dependencies, into a minimal Solidity excerpt. That excerpt is the scope of the analysis. If a function or library the property depends on is absent from the codebase, or otherwise unavailable to the agent, the agent stops and reports it.

`/right-round` builds the encoding as a Python program using Z3's Python bindings. The contract's state and the transaction's inputs become integer free variables, and the arithmetic that determines the property is defined as constraints over them. The property to verify is expressed as one or more predicates over those expressions. The solver searches for an assignment that satisfies them, returning `sat` with a witnessing assignment or `unsat` when none exists within the stated bounds. A condition that should be unreachable is asserted *directly*: a satisfying assignment is a witness that it can occur, and `unsat` rules it out within the bounds. A property that should hold in every reachable state, such as that a positive deposit always mints at least one share, cannot be posed this way; `/right-round` instead searches for a counterexample by asserting the *negation* of the property, so `sat` returns the values that break the property and `unsat` means it holds within the bounds. Figure 5.2 illustrates the counterexample search with a minimal encoding.

An ERC-4626 vault holds a reserve of one token and issues *shares*, transferable claims on that reserve: a user *deposits* assets and is minted shares in proportion to the assets the vault already holds, and later *redeems* shares back for assets (chapter 2). Both conversions use integer division, so they round.

We want to verify that a positive deposit always mints at least one share. Figure 5.1 shows a vault's `deposit`: it checks an access guard, mints shares one-to-one on the first deposit and otherwise as $\lfloor \text{assets} \cdot \text{totalShares} / \text{totalAssets} \rfloor$, transfers the tokens in, records the new shares, and emits an event. Only the share computation is relevant to this property; the guard, the transfer, the mint, and the event do not affect it. `/right-round` therefore encodes that computation and abstracts away the rest.

```
function deposit(uint256 assets, address receiver)
  external returns (uint256 shares)
{
  require(assets <= maxDeposit(receiver)); // ignored
  shares = totalShares() == 0
    ? assets // first deposit, 1:1
    : assets * totalShares() / totalAssets(); // kept
  token.transferFrom(msg.sender, address(this), assets); // ignored
  _mint(receiver, shares); // ignored
  emit Deposit(msg.sender, receiver, assets, shares); // ignored
}
```

Figure 5.1: A vault deposit.

The encoding (Figure 5.2) declares the deposited amount `assets` and the two vault totals as free variables, each an integer standing for an EVM machine word. It bounds each to its Solidity type, a ≥ 0 lower bound and an upper bound at the type width. These bounds serve a double purpose, fidelity and solver tractability: a tighter range gives the solver a smaller search space, which can make the instance easier.

The encoding follows the two cases of `deposit`. The floor division is stated by its quotient-remainder identity: the auxiliary variables `q` and `rem`, declared with the division they encode, satisfy $q \cdot \text{totalAssets} + \text{rem} = \text{assets} \cdot \text{totalShares}$ with $0 \leq \text{rem} < \text{totalAssets}$. It then asserts the negation of the property, that a positive deposit mints none. Z3 returns `sat`: when a share is worth more than one asset (say `totalAssets = 2`, `totalShares = 1`), a one-asset deposit rounds to zero shares.

```

from z3 import Int, Solver, If, Implies, And, Not

# --- free variables: contract state and transaction input ---
assets      = Int("assets")
total_assets = Int("total_assets")
total_shares = Int("total_shares")

s = Solver()

# --- bounds on the free variables: uint256 words, in [0, 2**256) ---
s.add(assets >= 0, total_assets >= 0, total_shares >= 0)
s.add(assets < 2**256, total_assets < 2**256, total_shares < 2**256)

# --- contract encoding: the deposit share computation ---
# a later deposit rounds down: q = floor(assets * total_shares / total_assets),
# by the quotient-remainder identity. Imposed only when total_shares > 0.
q  = Int("q")      # quotient of the ratio division
rem = Int("rem")    # its remainder
s.add(Implies(total_shares > 0,
               And(q * total_assets + rem == assets * total_shares,
                   rem >= 0, rem < total_assets)))
# deposit mints 1:1 on the first deposit, otherwise the rounded ratio:
shares = If(total_shares == 0, assets, q)

# --- property: a positive deposit mints >= 1 share (assert its negation) ---
s.add(assets >= 1, Not(shares >= 1)) # positive deposit that mints 0 shares
print(s.check())                     # sat: assets=1, TS=1, TA=2 -> 0 shares

```

Figure 5.2: A minimal SMT encoding using Z3 API

In Figure 5.2 the floor division is written out explicitly. That same quotient-remainder pattern recurs in every encoding. The operations that build on it recur too, such as the multiply-then-divide (`mulDiv`) operations and the share-to-asset conversions. Rather than re-derive these on each run, the skill provides them as a small, trusted library of arithmetic primitives, `fixedpoint.py`. An encoding imports the primitives it requires, such as `mul_div_floor` and `mul_div_ceil`, and is composed from them; the mint of Figure 5.2, for instance, reduces to a single `mul_div_floor` call. Defining the arithmetic once, in a reviewed library, eliminates a source of encoding error, renders each operation's rounding direction explicit and consistent across models, and preserves the readability of the encoding.

An SMT solver decides the satisfiability of a quantifier-free formula, which is finite and loop-free. A loop becomes such a formula only by unrolling it a fixed number of times, and unrolling it in full is either impossible, when the iteration count is not fixed in advance, or too large to solve.

/right-round therefore caps the unrolling at three iterations, reasoning only about executions that stay within that bound.

5.2 The Faithfulness Check

The faithfulness method of subsection 4.3.2 fixes the same inputs in both the encoding and the real contract, then checks that the output the solver derives from the encoding matches the contract's. The check is a bounded fuzz test, written in Solidity; Figure 5.3 shows it for the vault mint of Figure 5.2. Each iteration reads the vault's current totals, deposits a fuzzed amount through the real deposit function, and records the shares the vault mints. An evaluator script then imports the encoding, pins its free variables, the deposited amount and those totals, to the same values, solves, and returns the shares that satisfy the encoding. The test asserts that the two share counts agree; a mismatch on any input fails the test, and the agent must correct the encoding.

```
function testFuzz_deposit(uint256 seed, uint256 assets) public {
    // a first deposit, so the vault already holds a balance
    bound(seed, 0, 1e30);      // clamping
    _deposit(alice, seed);

    uint256 totalShares = vault.totalSupply(); // pre-state
    uint256 totalAssets = vault.totalAssets();

    // the deposit under test, through the real deposit()
    bound(assets, 0, 1e30);    // clamping
    uint256 shares = _deposit(bob, assets);

    // compare against the encoding evaluated at the same values
    uint256 z3 = callEncoding(assets, totalShares, totalAssets);
    assertEq(shares, z3, "shares mismatch");
}
```

Figure 5.3: The faithfulness fuzz test for the vault mint of Figure 5.2.

A passing fuzz test is not, on its own, evidence of faithfulness: it tests the encoding only in the states that it reaches. The rounding in the Figure 5.1 deposit function runs only once the vault holds a balance, as the first deposit into an empty vault mints one-to-one and has no ratio division to round. A test that deposits and compares but never establishes a prior balance therefore stays on that first-deposit path, where floor and ceil coincide, so an encoding that rounds the wrong way matches the contract on every input and passes. Reaching a state where the rounding matters takes a deposit to establish outstanding shares. Mutation testing addresses this. A *mutation* is a copy of

the encoding with one arithmetic primitive's rounding flipped, from rounding up to rounding down or the reverse. The workflow builds one mutation per rounding operation in the encoding, so the mint encoding of Figure 5.2 yields a single mutation that rounds its division up instead of down, and runs the fuzz test against each. If the test passes on any mutation, it is too weak and the agent must strengthen it. The agent runs the solver only once the test fails on every mutation.

5.3 The Proof of Concept

When the verdict is SAT, `/right-round` turns the counterexample into a runnable proof of concept. The solver returns concrete values for the free variables (for the deposit property, `total_assets = 2`, `total_shares = 1`, and `assets = 1`). Even given the right values, producing a proof of concept by hand is time consuming. A sequence of operations that drives the real contract into the state that violates the property is the ultimate proof that the bug is real, increasing the analyst's trust in the result. `/right-round` therefore writes a Solidity test (Figure 5.4) that constructs this sequence.

```
function test_smtCounterexample_zeroShares() public {
    // reach the witness state: total_assets = 2, total_shares = 1
    _deposit(alice, 1); // 1:1 first deposit -> TS = 1, TA = 1
    token.transfer(address(vault), 1); // donation -> TA = 2, TS unchanged

    assertEq(vault.totalSupply(), 1);
    assertEq(vault.totalAssets(), 2);

    // deposit under test mints floor(1 * 1 / 2) = 0 shares, violating the property
    uint256 shares = vault.deposit(1, bob);
    assertEq(shares, 0, "positive deposit minted zero shares");
}
```

Figure 5.4: The generated PoC for the vault-mint counterexample.

5.4 The Artifacts

A run leaves a set of artifacts in a per-case directory (Figure 5.5). `prompt.md` contains the invocation verbatim; `excerpt.sol` is a minimal excerpt of the verified path; `encoding.py` contains the SMT encoding; `eval.py` is the evaluator; `mutations/` contains one file per flip-able call site; and `results.md` is the report. The Solidity tests are named after the property, such as `RightRound<PropertyName>.t.sol`, and live in the codebase's `test/` directory so that they run against the real contracts. Where the codebase is not a native Foundry project, they go in a `smt/` directory instead.

```

right-round/
|-- erc4626-share-mint/
|   |-- prompt.md           # the /right-round invocation, verbatim
|   |-- excerpt.sol        # extracted Solidity
|   |-- encoding.py        # the Z3 encoding
|   |-- fixedpoint.py      # primitive library
|   |-- eval.py            # faithfulness evaluator
|   |-- mutations/         # one mutated encoding per rounding site
|       |-- step1-mul-div-ceil.py
|       |-- step2-mul-div-ceil.py
|       |-- step3-mul-div-ceil.py
|   |-- results.md         # the verification report

test/                      # in the codebase, so it runs against the real contracts
|-- erc4626-share-mint/
|   |-- RightRoundErc4626ShareMint.t.sol # faithfulness, mutation, PoC (FFI)

```

Figure 5.5: A `/right-round` run directory, one per verified property.

`results.md` follows a six-part structure: the verdict heading; a statement of what was verified, in English and in formal terms; a counterexample walkthrough, or for an UNSAT the argument for why the property holds, with the lemma chain when one was used; the encoding design as a free-variable table together with the protocol invariants; the exact commands to reproduce each result with its expected outcome; and a table mapping each file to its purpose.

5.5 The Plugin

`/right-round` is packaged as a plugin for coding agents, bundling three skills: the `right-round` verification skill and two companions, `anchor` and `bounds`. Figure 5.6 shows its layout.

```

right-round/
|-- skills/
|   |-- right-round/           # the verification skill
|   |   |-- SKILL.md           # procedure + encoding conventions
|   |   |-- fixedpoint.py      # primitive library
|   |   |-- encoding_template.py # encoding skeleton
|   |   |-- eval_template.py    # evaluator skeleton
|   |   |-- excerpt.py         # excerpt/source consistency checker
|   |   |-- examples/          # artifacts from example runs
|-- anchor/
|   |-- SKILL.md
|-- bounds/
|   |-- SKILL.md

```

Figure 5.6: The `/right-round` plugin: a manifest and three skills.

The procedure and the encoding conventions the agent follows are set out in `SKILL.md`. Alongside it the skill ships the primitive library `fixedpoint.py` (section 5.1), the encoding and evaluator skeletons `encoding_template.py` and `eval_template.py`, and `excerpt.py`, which checks a written excerpt against the source ranges it cites. Shipping these as skeletons fixes the structure and the interfaces the rest of the workflow relies on. The evaluator exposes a fixed command-line interface that the faithfulness fuzz test invokes; the same script drives every mutation by pointing it at the mutated encoding, so the main model and its mutations are all exercised through one interface. The encoding skeleton confines the agent’s edits to a few marked sections. Together this keeps artifacts comparable across runs and easier to inspect and trust. The `examples/` directory ships the artifacts of a few example runs: each is a case directory of the same shape as Figure 5.5, holding a property with its excerpt, encoding, evaluator, mutations, and report. The agent reads these as references to produce its own artifacts in the same structure. The agent loads `SKILL.md` when the skill is invoked and reads the other files as the steps need them.

The two companions act on an artifact directory `right-round` has already produced. `anchor` turns a counterexample into a proof of concept that runs against the contract’s live on-chain state. The analyst provides a small configuration file, `chain.env`, giving the target chain, the block to read at, an RPC endpoint, and the root contract. `anchor` generates a shell script of RPC calls that the analyst runs to fetch the values at that block: it groups variables sharing a getter into one call, resolves the addresses of the contracts that are part of the system through their getters, and skips any variable whose source does not resolve to a public getter. `anchor` then pins each fetched value into the encoding and re-solves. `bounds` lets the analyst inspect and adjust the range bounds on the free variables and re-solve (subsection 4.3.3). The natural bound for each free variable is its Solidity type, but at full `uint256` width the solver may not return a verdict within a practical time. Narrowing the bounds restores tractability at the cost of coverage: an `unsat` under narrowed bounds

only rules out a counterexample within the chosen range. The range that is both solvable and large enough to be meaningful, such as the realistic magnitude of a token supply, is a judgment about the protocol and often cannot be inferred by reading the code. Sometimes the skill also finds a counterexample that violates the property only at amounts too small to be significant in practice, and it is then useful to raise the lower bounds so the solver searches only for counterexamples large enough to matter. We came to recognize the importance of this following advice from real-world experts. bounds leaves the choice to the analyst, letting them inspect the current ranges, widen or tighten them, and re-solve.

Chapter 6

Evaluation

This chapter answers the three research questions of chapter 1: how often `/right-round` returns the correct verdict with good evidence (RQ1), what reaching a verdict costs in time, tokens, and money (RQ2), what the limitations of this approach are (RQ3).

We first introduce the benchmark (section 6.1) and the experimental setup (section 6.2). We then introduce the judge that inspects the artifacts of a `/right-round` verification and reports whether they qualify as good evidence (section 6.3). The three research questions follow in section 6.4, section 6.5, and section 6.6.

6.1 The Benchmark

`/right-round` is designed to be used on complex real-world codebases. To evaluate it we built a benchmark composed of 22 popular production repositories, among them OpenZeppelin, Solady, Uniswap, Aave, Morpho, and Compound. Most are unmodified upstream repositories, pinned at the last commit at the time of writing. We also authored 8 hardening experiments on forks of OpenZeppelin and Solmate, each adding a candidate *mitigation* or changing a parameter. Their purpose is to test whether a proposed defense actually makes a property hold that the original contract violates. Table A.1 in the appendix lists every fixture with its pinned commit and case count.

A test case is composed of a prompt that states a property, an expected verdict, and a repository pinned at a defined commit. The benchmark contains 288 cases, divided in the four protocol categories of section 2.2: *vaults*, *rebasing tokens*, *lending markets* and *automated market makers*. Table 6.1 gives the composition of the benchmark.

Category	Repos	Fixtures	Cases	SAT	UNSAT
ERC-4626 vaults	5	14	135	48	87
Rebasing tokens	2	2	8	6	2
Lending	8	10	67	27	40
AMM/DEX	8	8	78	35	43
Total	22	33	288	116	172

Table 6.1: Composition of the benchmark by protocol category. SAT and UNSAT specify the expected verdict.

By performing audits of a variety of projects in the industry for multiple months, we obtained the necessary domain knowledge to form the ground truth for the benchmark. The majority of the expected verdicts were established by manual analysis of what `/right-round` produced.

6.2 Experimental Setup

The tests are driven by a Python script that reads a case definition, prepares the repository by switching to the commit defined by the case, and invokes the coding agent with the case prompt. When the test is over, the script archives the repository containing the artifact directory of Figure 5.5. Then it records the agent transcript, the verdict parsed from the report and some metadata.

The tests execute inside a Docker container, on a single dedicated host: an Intel Core i5-1145G7 (4 cores, 8 threads, 2.6 GHz base and 4.4 GHz maximum) with 64 GB of RAM and NVMe storage, running Ubuntu 24.04.4 LTS on kernel 6.8.0 and Docker 29.6.2. The agent runs headless with permission prompts disabled. Each session executes as a separate unprivileged user inside the container, keeping the agent sandboxed.

We report a single configuration, frozen before the final campaign and summarized in Table 6.2.

Component	Claude	Gemini
Coding agent	Claude Code 2.1.218	gemini-cli 0.53.0
Model	claude-opus-4-8	gemini-3.1-pro-preview
Reasoning effort	high	default
Solver	Z3 5.0.0.0	Z3 5.0.0.0
<code>/right-round</code> skill	0.7.4	0.7.4

Table 6.2: Experimental configuration. Only the Claude CLI exposes a reasoning-effort setting.

6.3 Measuring Evidence Quality

Measuring how often `/right-round` returns the expected verdict alone is an incomplete metric to understand its effectiveness. A simple program that returns always UNSAT would already score 59.7% in our benchmark. Furthermore, a correct verdict could be backed by a wrong answer: a SAT result with an unreproducible counter-example, or an UNSAT with an invalid proof. Having good and correct evidence is what ultimately helps the analyst to quickly understand and validate a result.

To tackle the problem of measuring `/right-round` evidence quality, we introduce the `/vet` skill. `/vet` analyzes the artifacts (Figure 5.5) produced by a `/right-round` run, by applying seven checks:

Verdict consistency The verdict agrees across the report and `/vet`'s own re-execution of the runnable artifacts.

Check posing The main proof asks the question the prompt asked, and the verdict comes from a single solver check.

Encoding fidelity Rounding directions and arithmetic steps match the source and the model uses the primitive library.

Free variables There is no bound or relation between the free variables that does not exist in the original code and was not requested by the prompt.

Real-contract gate The faithfulness check ran against the real contract, executing the full relevant call path; the mutation set covers every flip-able call site.

Counterexample reproduction For a SAT, the witness was reproduced end to end on the real contract.

Artifact completeness The deliverable set exists, and the excerpt cites every encoded function.

Each run receives a judgement and a detailed description of why a judgement was given. For instance, `clean` means the evidence is flawless. Table 6.3 lists judgement that appeared at least once in the results, giving a brief definition for each of them.

Using a model to grade model output follows the LLM-as-a-judge line of work [25, 41], and judges of this kind have been applied to software artifacts as well as to prose [42].

Every judge instance runs on `claude-opus-4-8` at high reasoning effort at `/vet` version 0.7.4.

Code	Meaning
<code>clean</code>	the evidence is flawless.
<code>disclosed-scope-envelope</code>	there are disclosed per-variable upper bounds below the full type.
<code>disclosed-faithful-reduction</code>	the test does not deploy the protocol or use its public call paths: it mocks, writes state directly, or calls library functions.
<code>disclosed-over-approximation</code>	a quantity the contract computes is left free, so the SMT encoding admits states the original contract do not.
<code>incomplete-excerpt</code>	the excerpt omits a function the verdict turns on.
<code>sat-by-fuzzing</code>	the solver returned unknown and the witness was found outside the model, then reproduced on the real contract.
<code>incomplete-mutation-coverage</code>	fewer mutations than flip-able rounding sites.
<code>raw-z3-division</code>	native Z3 division on the arithmetic path.
<code>unjustified-scope-restriction</code>	a constraint non-existent in the original contract and not provided by the analyst.
<code>misposed-check</code>	the main check answers a different question than the prompt asked.
<code>not-exercising-real-contract</code>	faithfulness check ran against a reimplementation
<code>vacuous-main-check</code>	the main check decides trivially.
<code>vacuous-mutation-gate</code>	mutation tests fail for a reason unrelated to the mutation.
<code>missing-counterexample-reproduction</code>	a SAT never shown to reproduce on the original contract.
<code>no-encoding-delivered</code>	no SMT evidence exists; the verdict comes from prose.
<code>property-drift</code>	the property checked diverges from its siblings.

Table 6.3: The `/vet` codes that occur in our runs.

Disclosure in the report influences the grading. A disclosed reduction, such as an upper bound below the type size on the free variables, leaves the analyst aware of potential limitations of the verification process.

6.4 Effectiveness (RQ1)

RQ1 asks how often `/right-round` returns the correct verdict and provides good evidence (chapter 1). Table 6.4 shows a comparison between the performance of Claude and Gemini across our benchmark. A run counts as *correct* when its verdict agrees with the case’s expected verdict, and as *correct & clean* when it is correct and `/vet` judges it `clean`.

Category	Cases	Claude		Gemini	
		Correct	Correct & clean	Correct	Correct & clean
ERC-4626 vaults	135	135 (100%)	122 (90.4%)	129 (95.6%)	10 (7.4%)
Rebasing tokens	8	8 (100%)	8 (100%)	8 (100%)	0 (0%)
Lending	67	66 (98.5%)	55 (82.1%)	59 (88.1%)	7 (10.4%)
AMM/DEX	78	65 (83.3%)	47 (60.3%)	54 (69.2%)	1 (1.3%)
Total	288	274 (95.1%)	232 (80.6%)	250 (86.8%)	18 (6.2%)

Table 6.4: Verdict correctness and clean verdict rate by category, Claude vs Gemini.

In our benchmark, Claude reports the expected verdict in 95.1% of the cases, and 80.6% of them also pass the artifact audit. Gemini, on the other hand, reports the expected verdict in 86.8% of the cases, but fails consistently the artifacts evaluation, passing only in 6.2% of them.

Code	Claude	Gemini
clean	239	18
disclosed-scope-envelope	23	9
disclosed-faithful-reduction	2	1
disclosed-over-approximation	7	3
incomplete-excerpt	4	48
sat-by-fuzzing	1	2
incomplete-mutation-coverage	1	21
raw-z3-division	0	3
unjustified-scope-restriction	9	28
misposed-check	0	3
not-exercising-real-contract	1	64
vacuous-main-check	1	32
vacuous-mutation-gate	0	22
missing-counterexample-reproduction	0	6
no-encoding-delivered	0	28
property-drift	0	0
Total	288	288

Table 6.5: Runs carrying each /vet code, by coding agent.

The two coding agents differ in how they fail. Table 6.5 gives the distribution of judgements for each coding agent. Claude fails mainly in the choice of scope to encode: 9 unjustified-scope-restriction, 23 disclosed-scope-envelope, and 7 disclosed-over-approximation account for roughly 80% of its failures. Their seriousness varies: introducing a constraint that the original code does not enforce is dangerous and can lead to invalid proofs. Bounding a free variable below its Solidity type shows partial evidence that the property checked holds for any state. Modeling a

superset of the states the contract can reach is not an issue when the verdict is UNSAT, since the property then holds for every configuration, but can lead to unreproducible SAT. None of these failures led to a wrong verdict in our benchmark, and when the verdict is SAT all of them are mitigated by the production of the runnable PoC.

Gemini’s failure modes are much more diverse. The coding agent fails in four recurring ways: incomplete steps, poor choice of scope, poor execution, and ignoring clear instructions. Gemini fails to deliver a complete excerpt in 48 cases, a complete mutation coverage in 21, a counterexample reproduction in 6, and an encoding in 28. The agent fails to choose the correct scope of the analysis mainly by introducing constraints that are not enforced in the real contract. It fails to exercise the real contract when checking the faithfulness of the encoding 64 times, and in 32 cases it builds a main check that decides trivially, never testing the property. Finally, it ignores clear rules, such as using the primitive library for arithmetic operations instead of Z3’s native division. Generally, Gemini does not provide good evidence for its analyses.

Category	Cases	UNKNOWN		Gemini no verdict
		Claude	Gemini	
ERC-4626 vaults	135	0	0	2
Rebasing tokens	8	0	0	0
Lending	67	0	1	4
AMM/DEX	78	12	15	2
Total	288	12	16	8

Table 6.6: Cases left undecided, by category and coding agent. UNKNOWN is a reported non-decision; the last column is the runs that produced no verdict at all, which only occurred with Gemini.

The effectiveness of `/right-round` depends on the complexity of the arithmetic. Nonlinear integer arithmetic is undecidable [29]. SMT solvers use heuristics to work with nonlinear constraints, but no complete decision procedure exists. Our benchmark shows that this is an issue for a handful of cases, where a set of exceptionally complex nonlinear constraints is combined. Table 6.6 shows the cases that `/right-round` cannot decide. Every one of Claude’s UNKNOWN verdicts falls in the AMM/DEX category. A property that fails consistently, in 8 of the 12 cases, is the following: “depositing liquidity, performing a swap, then withdrawing liquidity does not yield more than swapping directly”, or simply “liquidity-swap-roundtrip”. To check this property the encoding has to include three complex arithmetic paths: the liquidity deposit, the swap, and the liquidity withdrawal. The analysis fails on the concentrated-liquidity implementations (section 2.2), Uniswap V3 and V4 and KyberSwap Elastic, and on Aerodrome constant-product pool. However, `/right-round` manages to prove simpler variants of it, for example by pinning the fee to zero on Uniswap V3 and V4, and the full property on simpler implementations such as Uniswap V2.

The remaining 4 UNKNOWN cases are two properties on two implementations, KyberSwap Elastic

and PancakeSwap StableSwap. The first one is a swap round trip: “swapping a token for another and immediately back does not return more than the initial amount”. The second one is a swap split: “a single swap can return more than two consecutive swaps of the same total amount”. These properties involve a single complex arithmetic path, the swap, but that has to be encoded 2 and 3 times respectively. Both implementations add a further complexity multiplier: on KyberSwap Elastic the property is checked over a tick crossing, and on PancakeSwap StableSwap a swap is priced using an iterative solver whose two loops are each unrolled 4 times¹ in the encoding. `/right-round` decides both properties on KyberSwap Elastic when the swap stays within a single tick. On PancakeSwap StableSwap neither is decided, and the benchmark contains no simpler variant of them. Essentially, these cases suffer from the same SMT solver limitations as the eight cases of the previous paragraph.

Although we have all these unknown results, `/right-round` managed to successfully check liquidity-swap-roundtrip, the property with the highest rate of failures, on one of the codebases with the most complex nonlinear integer arithmetic of the benchmark: PancakeSwap StableSwap. In this run, the coding agent first tried to follow the skill by building the SMT encoding. It later realized that the set of constraints was intractable by the solver, and ran a simulation that found a violation. It then built the PoC that triggers the violation on the real contract. The `/vet` judges this run as `sat-by-fuzzing`. On another case, a Uniswap V3 liquidity-swap-roundtrip, the agent went one step further: after finding the witness outside the model it pinned the free variables in the encoding and ran the solver to confirm it. We believe this could be a valid strategy to tackle solver timeouts and could be included in `/right-round`.

In practice, `/right-round` works well when the coding agent is good enough. With Claude, four out of five runs are correct and provide good evidence of the analysis. The failures concentrate in a few modes, and on this benchmark only one of them hid a real bug: a single lending case reported UNSAT where a counterexample exists, on a main check that decided trivially. Gemini comes close to Claude in giving the correct verdict, but with a drastically lower quality of artifacts.

6.5 Cost (RQ2)

RQ2 asks what is the cost of running `/right-round` in terms of time, tokens, and money. Table 6.7 and Figure 6.1 show the data we gathered by running the two coding agents on our benchmark.

¹`/right-round` unrolls loops 3 times by default. In this case we overrode the default in the prompt.

Metric	Claude		Gemini	
	All	Shared	All	Shared
Runs	288	18	288	18
Total time	19m 26s (35m 2s)	13m 2s (15m 37s)	16m 27s (38m 59s)	16m 0s (22m 11s)
Solver time	30s (2m 2s)	3s (4s)	47s (2m 54s)	4s (4s)
Test time	3m 52s (9m 18s)	2m 23s (3m 41s)	6m 38s (14m 4s)	7m 50s (11m 40s)
Unattributed	14m 6s (24m 14s)	10m 41s (12m 39s)	8m 27s (20m 17s)	8m 3s (14m 50s)
Tokens per case	11.54 M $\pm$ 8.57 M	7.71 M $\pm$ 2.56 M	3.00 M $\pm$ 2.64 M	3.09 M $\pm$ 1.60 M
Cached tokens	96.0% $\pm$ 1.3%	95.8% $\pm$ 1.2%	78.3% $\pm$ 11.5%	83.2% $\pm$ 6.5%
Main-model share	100.0% $\pm$ 0.0%	100.0% $\pm$ 0.0%	99.4% $\pm$ 0.5%	99.2% $\pm$ 0.5%
Cost per case	\$9.48 $\pm$ \$5.81	\$6.81 $\pm$ \$1.86	\$1.72 $\pm$ \$1.27	\$1.69 $\pm$ \$0.91

Table 6.7: Cost of a verdict by coding agent, over all runs and over the 18 shared cases. Durations are mean with the 90th percentile in parentheses; the rest are mean $\pm$ standard deviation.

The *All* columns cover every run. The *Shared* columns cover the 18 cases both agents complete cleanly. Those are the ones that compare the two coding agents on the same work.

The *total time* is the duration of a `/right`-round analysis, from the agent invocation to its termination. The *solver time* is the total run time of the solver. The *test time* including the faithfulness check and the PoC production, plus the eventual compilation of the contracts. The solver time and the test time include the time spent running wrong artifacts the agent fixes later in the analysis, and account for both background and foreground usage. *Unattributed* time is the total time minus foreground tool time. *Main-model share* is the fraction of tokens spent on `claude-opus-4-8` and `gemini-3.1-pro-preview` respectively, the remainder goes to cheaper models on sub-tasks.

The total time is recorded by the script that orchestrates the tests. The solver, test time and unattributed time are inferred from the agent transcript of the run. The token consumption, main-model share are also computed from the transcript. The cost is computed by applying the published list price of each model to its own token counts, separately for input, output, cached reads, and cache writes. The two main models are priced per million tokens at \$5.00 input and \$25.00 output for `claude-opus-4-8` [6], and \$2.00 input and \$12.00 output for `gemini-3.1-pro-preview` [23]. Cached reads cost \$0.50 and \$0.20 respectively.

Running the solver does not take significant time in most of the cases of our benchmark. On the shared cases it averages 3 seconds for Claude and 4 for Gemini, against a total time of 13 minutes 2 seconds and 16 minutes. Gemini spends more than three times as long on testing, 7 minutes 50 seconds against 2 minutes 23 seconds. The rest is LLM inference: 10 minutes 41 seconds for Claude, four fifths of its run, and 8 minutes 3 seconds for Gemini, half of its own. That time goes to reading the code, building the encoding, and iterating against the faithfulness check.

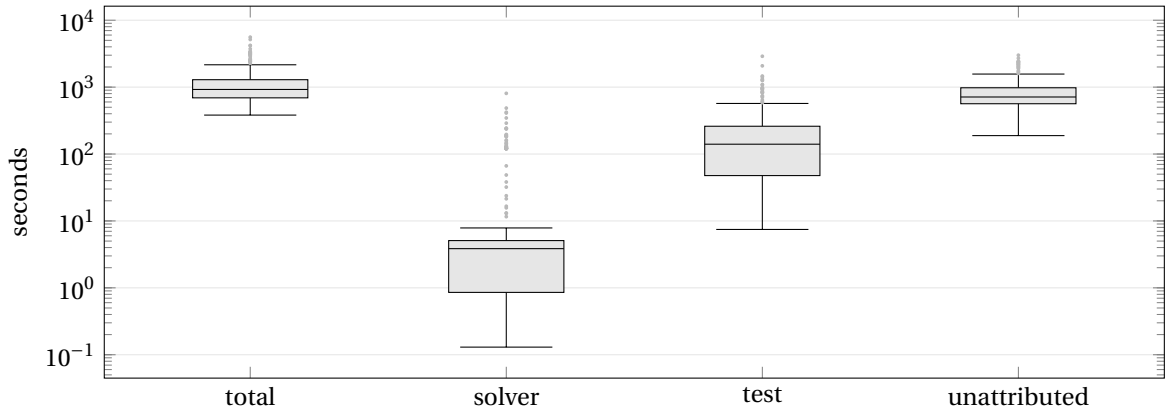


Figure 6.1: Time distribution of Claude’s four duration columns over all 288 runs. Boxes span the quartiles with the median inside. The scale is logarithmic. The two runs that spent no time in the solver are left out.

On the shared cases a `/right-round` analysis costs \$6.81 with Claude and \$1.69 with Gemini, spending 7.71M and 3.09M tokens on average. Over the whole benchmark Claude averages \$9.48 and 11.54M tokens per case. Both agents make good use of prompt caching, 96.0% of Claude’s tokens and 78.3% of Gemini’s being cache reads, which the two providers bill at a tenth of the fresh-input rate. At any plausible billing rate, \$9.48 is two to five minutes of an expert’s time and \$1.72 is under one, which is far below the time and cost needed to read the code, reason about it, or build an SMT encoding by hand.

6.6 Limitations (RQ3)

RQ3 asks what are the limitations of `/right-round`’s approach. This section identifies them in five classes, using evidence from the experiments of our benchmark.

6.6.1 Solver Intractability

A set of nonlinear integer arithmetic constraints can be too complex for the solver to decide (section 6.4). This happened on 12 of the 288 cases, all in AMM/DEX. It is a non-answer and not a wrong answer, and 7 of those runs are judged `clean` by `/vet`: the solver simply could not decide. Removing a nonlinear term restores tractability. Pinning the fee to zero decides the property on Uniswap V3 and V4, and restricting the swap to a single tick decides both swap properties on KyberSwap Elastic.

6.6.2 Bounded Loop Unrolling

A solver decides a quantifier-free formula, which cannot express loops. The standard practice is to unroll loops a fixed number of times, and `/right-round` defaults to three iterations, which a prompt can raise (section 5.1). Every property whose relevant arithmetic includes loops is checked only over executions that stay inside this bound. A SAT result keeps its validity, since a witness found inside the bound is a real execution, but an UNSAT verdict is proven only on a subset of all possible executions. In our benchmark, no bugs were missed due to this limitation. To make the analyst aware of this, `/right-round` discloses loop bounds in the final report.

6.6.3 Scope and Bounds

The agent decides what to encode and which bounds to put on the free variables. There is no mechanical check to verify these choices. This is the most frequent failure of our campaign. Nine of the Claude’s rejected runs carry a bound that the contract does not enforce and the analyst does not state.

`/right-round` requires every bound to be a type bound, a cited invariant, or a restriction defined by the analyst, and lists them in the report.

6.6.4 From Natural Language to Predicate

Two steps are unverified and rely on the quality of the coding agent: interpreting a property stated in natural language to find the contract functions it depends on, and translating it into a logical predicate. Underspecified prompts make this limitation more evident. In early versions of our benchmark, ambiguous prompts led to verdict flipping: two subsequent runs with the same prompts reporting different results. When the prompt is ambiguous, the coding agent can interpret it differently in different runs.

The report mitigates this, since the analyst can check that the property verified is the one they intended. Another mitigation works only when `/right-round` is run interactively: the agent can put an ambiguity back to the analyst as a clarifying question before it commits to a verification task. Our benchmark runs are headless (section 6.2), so every ambiguity is resolved by the agent’s judgement. Across the benchmark we carefully stated the property unambiguously.

6.6.5 Limitations of the Faithfulness Method

The faithfulness check fuzzes the SMT encoding against the compiled contract and reports agreement on every generated input. That is not a proof of equivalence. Hence, the SMT encoding could

be incorrect, leading to incorrect results. This limitation was not observed in our benchmark.

6.6.6 Overall

By evaluating `/right-round` on our benchmark, we found two main practical limitations. Complex sets of nonlinear integer arithmetic constraints are the only one that stops the implementation from producing an answer, on 12 of 288 cases. All of the undecided cases are in the AMM/DEX category. This limitation is inherited from the underlying SMT solver. A possible mitigation for this is using a fuzzer as a fallback, which has already proved its effectiveness on multiple runs of our experiments. This mitigation is further discussed in the future work section (section 8.1). The other limitation we observed is what the agent chooses as scope to encode and which bounds it picks for the free variables. We believe that this second point can be improved with skill optimizations and better coding agents.

Chapter 7

Real-World Case Studies

This chapter describes how `/right-round` was used at ChainSecurity, a smart contract auditing company.

7.1 Audit of a Morpho Integrator

Morpho is a popular lending market (section 2.2). ChainSecurity was auditing a Morpho integrator. It exposed a `maxBorrow` function, which computes how much a position can borrow without becoming unhealthy, i.e., failing the *health check*. The audit team wanted to verify a system requirement: that calling `borrow` with the value returned by `maxBorrow` always succeeds. The codebase shipped a fuzz suite, but no test covered this property.

Checking this by reading is hard because the call path has many complex arithmetic steps, which are hard for a human to take into account all at once. The property is about a chain of 4 divisions, each with their own rounding direction, a max function, and a min function, spread over two contracts (Figure 7.1).

$$(c, d, A, T, Y) \in [0, 2^{128}]^5, \quad p \in [1, 2^{128}), \quad \ell \in (0, W)$$

$$\begin{aligned}
b_{\text{pre}} &= \left\lceil \frac{d(A+1)}{T+V} \right\rceil && \text{debt in assets, before the borrow} \\
m &= \left\lceil \frac{\lfloor c p / O \rfloor \cdot \ell}{W} \right\rceil && \text{debt ceiling the health check allows} \\
a &= \min(\max(m - b_{\text{pre}}, 0), Y - A) && \text{amount maxBorrow reports} \\
\Delta d &= \left\lceil \frac{a(T+V)}{A+1} \right\rceil && \text{shares minted for } a \\
b_{\text{post}} &= \left\lceil \frac{(d + \Delta d)(A + a + 1)}{T + \Delta d + V} \right\rceil && \text{debt in assets, after the borrow}
\end{aligned}$$

$$\forall c, d, A, T, Y, p, \ell : \quad a > 0 \implies m \geq b_{\text{post}}$$

Here c is the position's collateral and d its debt in shares, A and T are the market's entire outstanding debt denominated in the loan asset and total debt shares issued across all borrowers, p is the oracle price, ℓ the integrator's conservative loan-to-value ratio, and Y the market's total supplied assets, of which $Y - A$ is free to be borrowed. The constants are the market's virtual-share offset V (section 2.4) and the fixed-point scales W of ℓ and O of p (section 2.3).

Figure 7.1: The arithmetic of `borrow(maxBorrow())`

The predicate we want to check is: whenever `maxBorrow` reports a positive amount a , borrowing exactly that amount leaves the resulting debt b_{post} within the ceiling m that the health check allows. `/right-round` asserts its negation, so `sat` returns a state in which the borrow reverts, and `unsat` means no such state exists within the bounds.

An early version of `/right-round` found a counterexample in about 45 minutes. Due to precision loss, the debt is inflated by one unit above what `maxBorrow` accounts for, leading to failing the health check. A fuzz test over the same property found no violation in 100 000 iterations. The finding was confirmed by the auditing team, the code was corrected by the developers, and valuable expert time was saved and spent securing different parts of the system.

Our benchmark (section 6.1) carries this property on the patched code. Today, `/right-round` proves it holds in about 21 minutes of unsupervised work.

7.2 Deeper Understanding of Rebasing Tokens

Rebasing tokens pay interest by automatically increasing holders' balances (section 2.2). Lido's stETH is one of the most popular ones. Every balance query and every transfer converts between shares and balances, and every conversion loses some precision. Implementations differ in how they handle that loss.

ChainSecurity wrote a blog article about them and their peculiar behavior, or "quirks" [13]. We first used `/right-round` to check the four properties the article states. Every verdict agreed with the article's analysis.

We continued our analysis asking whether these quirks could break some protocols integrating with rebasing tokens. One property was a particularly good candidate: on a transfer, stETH favors the sender. Hence, transferring an amount a can result in the recipient receiving $a - 1$.

We used `/right-round` to check the following property on different deployed integrators that allow deposits and withdrawals in stETH: depositing stETH and withdrawing it must not be profitable.

`/right-round` found violations in EigenLayer's `StrategyBaseTVLLimits` and by KelpDAO's `LRTDepositPool` with `LRTWithdrawalManager`. Both credit a deposit of X while receiving $X-1$. KelpDAO's deposit path shows the mechanism (Figure 7.2): the shares to mint are computed from the requested amount before any token moves, and the transfer that follows delivers one wei less than that. On both, `/right-round` reported that the round-trip property holds for a single deposit and is violated for two.

```
// LRTDepositPool.depositAsset
uint256 rsethAmountToMint =
    _beforeDeposit(asset, depositAmount, minRSETHAmountExpected);
IERC20(asset).safeTransferFrom(msg.sender, address(this), depositAmount);
_mintRsETH(rsethAmountToMint);

// LRTDepositPool.getRsETHAmountToMint, reached through _beforeDeposit
rsethAmountToMint =
    (amount * lrtOracle.getAssetPrice(asset)) / lrtOracle.rsETHPrice();
```

Figure 7.2: KelpDAO's stETH deposit.

The counterexamples did not match the live on-chain state, so we pinned some free variables by hand to let the solver find values that triggered the violation on-chain. This motivated the building of the `/anchor` companion skill (subsection 4.3.3).

The violation extracts value from the other depositors, as their shares are backed by less assets. Both findings were low severity, disclosed with a proof of concept, and acknowledged by the teams, which assessed no significant funds at risk.

7.3 Performance in Case Studies

Domain experts found `/right-round` useful on real code. It uncovered previously unknown bugs, and it has been used in further ChainSecurity engagements not described here.

Chapter 8

Conclusion

In this thesis we designed and implemented `/right-round`. `/right-round` takes a safety property stated in natural language and a codebase and determines whether the property holds. Alongside, it delivers helpful evidence supporting its verdict. We researched its effectiveness, cost and limitations.

The effectiveness heavily depends on the coding agent used. On our benchmark, Claude Code returns the expected verdict on 95.1% of the cases, and on 80.6% it provides flawless evidence of its validity. On 4.2% of the cases the agent does not report a verdict. Only on 0.7% of the cases the agent reports an incorrect verdict. Gemini returns the expected verdict on 86.8% of the cases, but provides flawless evidence in only 6.2% of the cases. Proving that with a good coding agent, `/right-round` can be used on real-world security reviews.

A `/right-round` analysis takes an average of 19 minutes of unsupervised work. Producing this analysis manually takes more time. Therefore, the time needed is acceptable to be used in production.

Using Claude Code, the cost of an average run is \$9.48. Far below the cost of employing an expert to perform the analysis. Hence, making it economically practical to be used in industry.

We found the main limitation to be properties and codebases that result in complex constraints over nonlinear integer arithmetic. When this is the case, `/right-round` does not return a verdict. This results in little time lost and does not return incorrect results. Therefore, this limitation carries little harm in practice.

Based on the outcome of these research questions, we believe `/right-round` partially addresses an existing security weakness in the DeFi ecosystem.

8.1 Future Work

In this section we explore ideas for future work.

8.1.1 The Solver as Confirmer

The evaluation chapter shows that some runs benefit from using guided fuzzing as a fallback. When the solver cannot handle too complex constraints, the agent notices it could try to use guided fuzzing on a small Python implementation of the complex arithmetic. The agent uses this to find a violation, then uses the SMT encoding to quickly confirm its validity. The counterexample was then reproduced end to end on the real contract.

The solver took a new “confirmer” role. This mode is worth exploring to make deliberate: on a solver timeout, direct the agent to fallback to fuzzing.

8.1.2 Other Source Languages

`/right-round` is optimized for Solidity by including in the skill domain knowledge and examples of to use the testing framework. The SMT encoding is a Python program that could in principle model any program. The implementation could be easily adapted to another programming language, as long as a testing framework similar features is available.

Bibliography

- [1] a16z. *Halmos: A Symbolic Testing Tool for EVM Smart Contracts*. <https://github.com/a16z/halmos>. 2025.
- [2] Hayden Adams, Noah Zinsmeister, and Dan Robinson. *Uniswap v2 Core*. <https://app.uniswap.org/whitepaper.pdf>. 2020.
- [3] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. *Uniswap v3 Core*. <https://app.uniswap.org/whitepaper-v3.pdf>. 2021.
- [4] Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. “SolCMC: Solidity Compiler’s Model Checker”. In: *Computer Aided Verification (CAV 2022)*. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 325–338. DOI: 10.1007/978-3-031-13185-1_16.
- [5] Anthropic. *Extend Claude with skills*. <https://code.claude.com/docs/en/skills>. Claude Code documentation. 2025.
- [6] Anthropic. *Pricing*. <https://docs.claude.com/en/docs/about-claude/pricing>. Accessed 20 August 2026. 2026.
- [7] Clark Barrett and Cesare Tinelli. “Satisfiability Modulo Theories”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.
- [8] Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas. “Towards Benchmarking of Solidity Verification Tools”. In: *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Vol. 118. OASICS. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 6:1–6:15. DOI: 10.4230/OASICS.FMBC.2024.6.
- [9] Paul Razvan Berg. *PRBMath: Solidity Library for Advanced Fixed-Point Math*. <https://github.com/PaulRBerg/prb-math>. 2025.
- [10] Bunni. *Exploit Post-Mortem*. <https://blog.bunni.xyz/posts/exploit-post-mortem/>. 2025.
- [11] Certora. *Prover Approximations*. <https://docs.certora.com/en/latest/docs/prover/approx/index.html>. 2026.

- [12] Certora. *wala-solidity: WALA-Based Solidity Static Analysis (RoundAbout)*. <https://github.com/Certora/wala-solidity>. 2025.
- [13] ChainSecurity. *The Hitchhiker’s Guide to Rebasing Tokens (stETH & aTokens)*. <https://www.chainsecurity.com/blog/the-hitchhikers-guide-to-rebasing-tokens>. 2026.
- [14] Stefanos Chaliasos, Marcos Antonio Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitrios Mitropoulos, and Benjamin Livshits. “Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?” In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE ’24)*. ACM, 2024. DOI: 10.1145/3597503.3623302.
- [15] Check Point Research. *How an Attacker Drained \$128M from Balancer Through Rounding Error Exploitation*. <https://research.checkpoint.com/2025/how-an-attacker-drained-128m-from-balancer-through-rounding-error-exploitation/>. 2025.
- [16] ConsenSys. *Mythril: A Symbolic-Execution-Based Security Analysis Tool for EVM Bytecode*. <https://github.com/ConsenSysDiligence/mythril>. 2025.
- [17] Crytic (Trail of Bits). *Medusa: Parallelized, Coverage-Guided, Mutational Solidity Smart Contract Fuzzing*. <https://github.com/crytic/medusa>. 2025.
- [18] Crytic (Trail of Bits). *roundme: A Human-Assisted Rounding Analyzer*. <https://github.com/crytic/roundme>. 2025.
- [19] DefiLlama. *Total Value Locked in DeFi*. <https://defillama.com/>. Accessed 19 August 2026. 2026.
- [20] Michael Egorov. *StableSwap: Efficient Mechanism for Stablecoin Liquidity*. <https://classic.curve.finance/files/stableswap-paper.pdf>. 2019.
- [21] Josselin Feist, Gustavo Grieco, and Alex Groce. “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, 2019, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008.
- [22] Foundry. *Foundry: A Portable and Modular Toolkit for Ethereum Application Development*. <https://github.com/foundry-rs/foundry>. 2025.
- [23] Google. *Gemini Developer API Pricing*. <https://ai.google.dev/gemini-api/docs/pricing>. Accessed 20 August 2026. 2026.
- [24] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. “Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’20)*. ACM, 2020, pp. 557–560. DOI: 10.1145/3395363.3404366.

- [25] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. *A Survey on LLM-as-a-Judge*. 2025. arXiv: 2411.15594. URL: <https://arxiv.org/abs/2411.15594>.
- [26] Hacken. *Web3 Security Report 2025*. <https://hacken.io/insights/2025-security-report/>. 2026.
- [27] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. “KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217. DOI: 10.1109/CSF.2018.00022.
- [28] Lido. *Lido Tokens Integration Guide*. <https://docs.lido.fi/guides/lido-tokens-integration-guide/>. 2026.
- [29] Yuri V. Matiyasevich. “Enumerable Sets are Diophantine”. In: *Soviet Mathematics. Doklady* 11.2 (1970), pp. 354–358.
- [30] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.
- [31] OpenZeppelin. *ERC-4626*. <https://docs.openzeppelin.com/contracts/5.x/erc4626>. 2025.
- [32] OpenZeppelin. *OpenZeppelin Contracts*. <https://github.com/OpenZeppelin/openzeppelin-contracts>. 2025.
- [33] rekt.news. *zkLend: Rekt*. <https://rekt.news/zklend-rekt>. 2025.
- [34] Runtime Verification. *Kontrol: Formal Verification of Foundry Tests with KEVM*. <https://github.com/runtimeverification/kontrol>. 2025.
- [35] Joey Santoro, t11s, Jet Jadeja, Alberto Cuesta Cañada, and Señor Doggo. *EIP-4626: Tokenized Vaults*. <https://eips.ethereum.org/EIPS/eip-4626>. 2021.
- [36] Vectorized. *Solady: Optimized Solidity Snippets*. <https://github.com/Vectorized/solady>. 2025.
- [37] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: Token Standard*. <https://eips.ethereum.org/EIPS/eip-20>. 2015.
- [38] Sam Werner, Daniel Perez, Lewis Gudgeon, Aariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. “SoK: Decentralized Finance (DeFi)”. In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT ’22)*. ACM, 2022, pp. 30–46. DOI: 10.1145/3558535.3559780.
- [39] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. <https://ethereum.github.io/yellowpaper/paper.pdf>. Ethereum Project Yellow Paper. 2014.

- [40] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *The Eleventh International Conference on Learning Representations (ICLR 2023)*. arXiv:2210.03629. 2023.
- [41] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023. arXiv: 2306.05685. URL: <https://arxiv.org/abs/2306.05685>.
- [42] Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luis F. Gomes, Guang Yang, Kui Liu, Xin Xia, and David Lo. *An LLM-as-Judge Metric for Bridging the Gap with Human Evaluation in SE Tasks*. 2025. arXiv: 2505.20854. URL: <https://arxiv.org/abs/2505.20854>.

Appendix A

Benchmark Fixtures

Table A.1 lists every fixture in the benchmark of section 6.1: the repository, the commit it is pinned at, a label for the variant where one repository contributes several commits, the number of cases pinned against it, and how those cases split between expected SAT and expected UNSAT. The contents are derived from the case manifests by the benchmark’s own inventory tool, and regenerated whenever the benchmark changes.

Repository	Commit	Variant	Cases	SAT / UNSAT
aave-v3-core	782f51917056		9	5 / 4
aave-v3-origin	6138e1fda458		13	8 / 5
aerodrome	1ba30815bba6		7	4 / 3
biswap-core	78a67b2bf9dc		7	2 / 5
bold	c8a5a4ee2e9d		8	2 / 6
comet	f766f51583c2		10	3 / 7
euler-vault-kit	5b98b42048ba		9	3 / 6
ks-elastic-sc	4ab08c0a60f7		12	8 / 4
lido-core	96738395ca3b		4	3 / 1
morpho-blue	528ccf43568e		8	1 / 7
morpho-integrator	12bd55140220		3	1 / 2
morpho-integrator	7056bb17257b	deposit-LTV bound	1	1 / 0
morpho-integrator	cb50ff25b9d6	burn share offset	1	1 / 0
openzeppelin-contracts	0a25c1940ca2	v4.8.3	10	4 / 6
openzeppelin-contracts [†]	23f7ddb14e92	minimum-shares check	9	3 / 6
openzeppelin-contracts	5fd1781b1454	v5.6.1	9	3 / 6
openzeppelin-contracts [†]	9602c7627ee8	max-share-price check	9	3 / 6
openzeppelin-contracts [†]	bae1a97cbaf6	virtual shares 1 to 10	9	4 / 5
openzeppelin-contracts [†]	c7fb9d9d996a	_decimalsOffset() = 1	9	1 / 8
openzeppelin-contracts [†]	fb848e4ef28f	zero-shares guard, 10x supply	10	3 / 7
openzeppelin-contracts [†]	fc2c23a018da	zero-shares guard	10	4 / 6
pancake-smart-contracts	cb079908a303		8	7 / 1
pancake-swap-core	3b214306770e		7	2 / 5
solady	acd959aa4bd0	v0.1.26	9	3 / 6
solmate [†]	a94c19b5cd74	minimum-shares check	10	4 / 6
solmate [†]	b20e136a4124	max-share-price check	10	4 / 6
solmate	c93f7716c990	v6.8.0	10	4 / 6
sparklend-v1-core	8120e495061d		9	5 / 4
tokenized-strategy	82806289f967		12	5 / 7
v2-core	6a9e7c978606		7	2 / 5
v3-core	d0831dc6b8a3		15	5 / 10
v4-core	46c6834698c4		15	5 / 10
vault-v2	ffa22a60ce14		9	3 / 6
33 fixtures, 22 repositories			288	116 / 172

Table A.1: Benchmark fixtures. A *fixture* is a repository at one pinned commit; a repository contributing several commits appears once per commit. [†] marks the eight fixtures whose commit is a variant we authored, each adding a candidate mitigation or changing a parameter (section 6.1); every other commit is unmodified upstream. v2-core, v3-core, and v4-core are Uniswap; vault-v2 is Morpho; bold is Liquity.