
# Metamorphic Testing for Floating-Point Performance Issues in SMT Solvers

Rosa Abbasi[1] (✉) and Eva Darulova[2]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany, rosaabbasi@mpi-sws.org
[2] Uppsala University, Uppsala, Sweden, eva.darulova@it.uu.se

**Abstract.** SMT solvers are essential in various domains, including program verification and synthesis. Although their correctness and performance have been extensively studied, performance testing for the floating-point theory remains limited, particularly for real-world queries. We propose a metamorphic testing approach that uses semantics-preserving rewrite rules, focusing on floating-point special values and peep-hole optimizations, to uncover meaningful performance issues in SMT solvers' handling of floating-point formulas. Using real-world test inputs, our approach is able to identify for every solver tested SMT queries for which solving time increases when the queries are simplified; we see slowdowns of up to 33.4x for Z3, 5.6x for MathSAT, 4.5x for cvc5, and 1.8x for Bitwuzla. We further observe that the approach is less successful when using input files that were randomly generated from a grammar.

## 1 Introduction

For the last few decades, SMT solvers have been used in many domains, such as program verification [3,11], symbolic execution [8,13], and program synthesis [4]. The efficiency and correctness of tools that use SMT solvers highly depend on the solver's efficiency and correctness, so the solvers have been subjected to extensive correctness testing [33,18,31,34,23,29,32] and, on fewer occasions, performance testing [27,30]. Given the complexity and scale of modern solver implementations, SMT solvers are typically treated as black boxes during testing, with issues being detected and reported to solvers' developers rather than debugged at the implementation level. As a result, SMT solvers have significantly improved over time across various theories.

However, performance on queries over the floating-point theory remains expensive and unpredictable; we observed unexpected performance behavior in our prior work [2]. This is unfortunate, as automated verification support for floating-point arithmetic specifically is highly valuable as it is unintuitive to many developers. For example, reasoning about floating-point special values (Not-a-Number (`NaN`), infinity, or signed zeroes) is challenging and often misunderstood [12].

To illustrate an instance of unexpected performance behavior, consider the partial SMT query shown in Listing 1.1 that we extracted from one of the verification conditions (VCs) generated by the KeY deductive verifier [3] for a real-world Java program. The property that KeY checks in this case is that scaling

a rectangle does not result in NaN nor infinity. To keep the listing brief, we exclude the remaining assertions covering other verification-condition branches, as well as the variable definitions and assertions regarding their range constraints. Z3, cvc5, MathSAT, and Bitwuzla require 9.37, 1.98, 3.4, and 0.25 seconds (average across 5 runs), respectively to solve this SMT query and produce unsat.

Listing 1.1: Partial SMT Query generated by KeY

```
1  ; additional details elided, incl. variable declaration and range constraints
2  (assert (fp.leq  (fp.sub RNE  (fp.add RNE  (fp.add RNE
3  (fp.mul RNE  |field_benchmarks.rectangle.Rectangle::$x| ui_arg1)
4  (fp.mul RNE  |field_benchmarks.rectangle.Rectangle::$y|
5  (fp #b0 #b0...0 #b00...00))) (fp #b0 #b0...0 #b00...00)) ; constants 0.0 (zeroes elided)
6  (fp.add RNE  (fp.add RNE  (fp.mul RNE
7  (fp.add RNE  |field_benchmarks.rectangle.Rectangle::$x|
8  |field_benchmarks.rectangle.Rectangle::$width|) ui_arg1)
9  (fp.mul RNE  |field_benchmarks.rectangle.Rectangle::$y|
10 (fp #b0 #b0...0 #b00...00))) (fp #b0 #b0...0 #b00...00))) (fp #b0 #b0...0 #b00...00)))
```

We now *simplify* the multiplication terms on lines 4 and 9. We use the floating-point identity twice that if the first operand in a multiplication is negative and neither NaN nor infinity and the second operand is either $+0.0$ or $-0.0$, then the multiplication can be replaced by the negation of the second operand:

$$u_1 \neq (\texttt{NaN} \vee \pm\infty) \ \wedge \ u_1 < 0 \ \wedge \ (u_2 = +0.0 \vee u_2 = -0.0) \Rightarrow (u_1 \times u_2) \longrightarrow -u_2$$

Having simplified the query, we would expect improved solver performance. This is, however, only the case for Z3 and cvc5, whose solving times improve by 22% and 7% (7.30s, 1.84s), respectively. For MathSAT and Bitwuzla, the solving time increases by 28% and 40% (to 4.37s and 0.35s). We are specifically interested in slowdowns in solving time and call them performance issues.

Our goal is to identify such **performance issues** in the implementation of the floating-point theory across various SMT solvers on real-world and grammar-generated benchmarks. We define a performance issue in this context as a statistically significant slowdown *of an individual solver* on a semantically equivalent, structurally simpler query, while acknowledging that such transformations may interact differently with solver heuristics and preprocessing.

Performance issues in floating-point reasoning affect multiple stakeholders. Identifying solver-specific performance issues can help SMT solver developers to debug and optimize their implementations. Tool developers who rely on SMT solvers can benefit from insights into how floating-point constraints can be effectively simplified before they are passed to SMT solvers, e.g., by optimizing VCs. End users of such tools may be discouraged from enabling or using floating-point verification support when it results in excessive and unpredictable runtimes, despite the much-needed robust floating-point support in verification tools [14].

Detecting such performance issues in SMT solvers is non-trivial, and has not been considered in existing testing approaches that primarily focus on correctness [33,18,31,34,23,29,32] rather than performance [27,30]. Those that consider performance record timeouts, or compare solving times between different

solvers, i.e., they rely on oracles or perform differential testing. Neither of those is well-suited to detect performance issues for an individual solver. Defining an oracle would amount to defining the expected solving time, which is not feasible. Comparing different solvers against each other evaluates a different metric; a longer runtime compared to different solvers *on an individual SMT file* does not necessarily indicate a performance issue in our sense, rather these individual differences may stem from different internal heuristics, implementation choices, or algorithms. While comparing different solvers (on the same inputs) is certainly important, we believe that identifying unexpected behavior of one solver on (highly) related inputs provides orthogonal and important insights.

We propose metamorphic testing to detect performance issues. Starting from an input SMT query using the floating-point theory, our approach applies up to two semantics-preserving rewrite rules (randomly chosen from a larger set), each of which aims to *simplify* the query—we would thus expect the query to be solved faster. By comparing individual solver performance on the original vs. rewritten queries, we can identify issues when the rewritten query takes longer to solve. Rule application is limited to a maximum of two rules per file to balance introducing meaningful syntactic variation with avoiding excessive simplification, which can mask performance issues by consistently producing speedups.[3]

As a secondary objective, we can compare performance sensitivity to these rewrite rules across solvers to further highlight the unpredictability of the floating-point theory implementations in SMT solvers. In contrast to prior metamorphic testing targeting incompleteness bugs [7], we use semantics-preserving rewrites to detect unexpected performance slowdowns within individual SMT solvers. While we focus on the floating-point theory due to its unpredictable performance and unintuitive nature, the approach itself generalizes to other SMT theories.

Our rewrite rules are drawn from common simplifications in floating-point reasoning and include rules targeting floating-point special values (Not-a-Number (NaN), infinities, signed zeroes), as well as rules derived from floating-point compiler (peep-hole) optimizations [24]. While many of these rules involve special values, which are known to be challenging for developers [12], all rules are semantics-preserving and aim to simplify floating-point expressions. Our implementation in the prototype tool GINGER applies such rewrite rules to SMT queries and compares solver performance before and after rewriting fully automatically.

An additional challenge, particularly for floating-point arithmetic, is that floating-point SMT queries stemming from real-world benchmarks are rare. Only few floating-point benchmarks in SMT-COMP have any special values, and when they do, those test files are usually quite small. As we show in this paper, automatically generating floating-point input files, e.g., from a grammar, with semantic and structural complexity of real-world problems is non-trivial, too.

We apply our approach to four SMT solvers (cvc5, Z3, MathSAT, Bitwuzla) using three benchmark suites: verification conditions from real-world programs, SMT queries (randomly) generated from a grammar, and mutations derived

[3] Once the identified performance issues are addressed, each rewrite rule on its own is expected to consistently improve solver performance.

from real-world verification conditions. Our results show that our metamorphic testing approach is effective in detecting performance issues. We are able to identify performance issues in every solver tested: we observe slowdowns on the simplified files of up to 33.4x for Z3, 5.6x for MathSAT, 4.5x for cvc5, and 1.8x for Bitwuzla. We also observe that realistic test inputs, along with mutants derived from them, are significantly more effective in exposing performance issues compared to grammar-generated test files.

As expected (and illustrated in the example above), the rewrite rules affect different solvers differently, i.e., simplifying (real-world) SMT queries improves solving time for some, but typically not for all tested solvers. Moreover, files that undergo more rewrite-rule applications tend to show larger improvements, highlighting the potential of rewriting—whether as a preprocessing step or within solver implementations—as a practical performance optimization.

*Contributions* In summary, our main contributions are:

- We present a metamorphic testing–based approach for detecting performance issues in SMT solvers on the floating-point theory (Section 2).
- We implement this approach in the tool Ginger (Section 4), which is released as open source[4]), together with all tested SMT files.
- We use Ginger to test four SMT solvers with support for the floating-point theory on three benchmark suites (Section 3), and identify performance issues (Section 5). We have submitted selected performance issues to the corresponding SMT solver developers. For cvc5, the reported performance issue was confirmed [1]; an update independent of this paper fixes this particular issue.

## 2 Metamorphic Testing for Performance Issues

Metamorphic testing addresses both the input generation and the oracle problem in software testing [9]. Given a known relationship between multiple inputs to a system, metamorphic testing defines the expected relationship between their corresponding outputs. A violation of this expected relation indicates a potential issue.

In our context, the relationship between the original and rewritten SMT formulas is one of semantic equivalence and simplification, achieved through the application of a set of semantics-preserving floating-point rewrite rules to the input SMT files. Since both versions represent the same logical problem and the rewrite rules are designed to *simplify* formulas, we expect solvers to produce identical results (`sat`/`unsat`) and the rewritten formula to have comparable or reduced solving time. Here, we do not use a formal notion of simplification; rather, simplification refers to transformations that reduce the number of nodes in SMT expressions and, from the perspective of users inspecting SMT queries, result in structurally simpler expressions.

[4] `https://github.com/rosaAbbasi/ginger`, artifact: `https://doi.org/10.5281/zenodo.21500107`

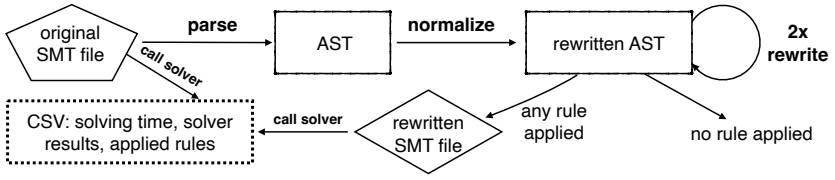



Fig. 1: Workflow of our metamorphic testing approach for performance issues

Metamorphic testing thus fits naturally for identifying performance issues in SMT solvers, as it enables evaluating each solver in isolation without the need for an external performance oracle or reference solver—both of which are difficult to define in this context.

Figure 1 illustrates the workflow of our metamorphic testing approach. Each SMT test input file is parsed into an abstract syntax tree (AST) and normalized slightly to ensure easier applicability of the rules. A randomly shuffled set of rewrite rules is applied sequentially to all `assert` statements (in all of our benchmarks, all relevant operations appear in `assert` statements) such that at most two rules are applied to a file. We limit the application to at most two rules per file to avoid oversimplifying formulas, which would obscure performance anomalies by causing only speedups. We observed empirically that using (at most) two rules works consistently well across our benchmarks. We additionally shuffle the rule order so that, under this limit, we vary which rules get applied.

Each `assert` statement is processed recursively, with applicable rules applied to any nested expression. Meaning, a rule may be applied multiple times within a file, depending on the number of matching terms. If at least one rule is applied to a test file, all selected solvers are executed five times on both the original and rewritten version of the file. We record all statistically significant relative differences in runtime between these two versions. Regressions, i.e. when the rewritten file is solved slower, indicate a potential performance issue.

Note that the order of rule application matters, as we impose a limit of at most two rule applications per file. In particular, the first two rules (possibly the same rule applied twice) whose preconditions are satisfied are applied, even though additional rules might also be applicable to the same file. Consequently, different rule application orders generate different transformed files. Moreover, applying one rule modifies the query, and depending on which rule is applied, the preconditions for another rule may no longer hold, thereby preventing its application. While the concrete transformations may differ depending on the order of rule application, we experimented with different rule orders and found that the overall trends observed in our large-scale experiments remain stable, with no qualitative differences in the results.

### 2.1 Normalization

Before applying the rewrite rules, we perform a syntactic normalization step to ensure that special-value handling is explicit and uniform across all benchmarks.

These transformations improve the consistency of subsequent rewriting and analysis steps and simplify the implementation. These transformations are themselves a form of rewrite rule application; thus, the term normalization is used informally here and should not be understood in the formal sense in term rewriting. This preprocessing pass is applied once and includes the following transformations:

1. Occurrences of values such as $+0.0$, $-0.0$, `NaN`, $-\infty$, and $+\infty$ that appear in assertions of the form `(assert (fp.eq [variable] [value]))` or `(assert (= [variable] [value]))` are replaced by substituting the variable with the corresponding constant throughout all assertions.
2. Additional assertions are introduced to explicitly state that a constant is not `NaN`, not infinite, and not zero, whenever its assigned floating-point value does not correspond to these special cases.
3. When a variable is constrained by upper and lower bounds—expressed through comparisons such as `fp.gt` or `fp.lt` with constant floating-point values— if applicable, new assertions are added to declare that the variable is not zero, not infinite, not `NaN`, and that it has the appropriate sign (positive or negative).
4. Equality checks such as `(=u (_ NaN [eb] [sb]))` or `(=(_ NaN [eb] [sb]) u)` are replaced with the semantically equivalent predicate `(fp.isNaN u)`.

### 2.2 Rewrite Rules

The rewrite rules are semantics-preserving transformations over the SMT-LIBv2 floating-point theory. Although expressed as rewrite rules, they are not used as a term-rewriting system and are not intended to reach a normal form. In total, our rule set consists of 51 rewrite rules that we derived from three sources:

I. the IEEE 754-2008 standard for floating-point arithmetic [15],
II. verified compiler optimizations from LifeJacket [24], and
III. additional rules we inferred for handling special values.

We collected the rules in group I directly from identities from the IEEE standard. For group II, we adapted rule implementations from the LifeJacket repository to the SMT context, ignoring rules that are not relevant to SMT formulas such as optimizations relying on the `nsz` flag, which ignores the sign of zero. Finally, we inferred several rules from recurring special-value checks observed in floating-point verification conditions. These rules simplify predicates such as `fp.isNaN`, `fp.isInfinite`, and `fp.isZero` when applied to compound expressions by exploiting semantic information about their operands. The rules are designed primarily for simplification, normalization, and redundancy reduction in `assert` statements. We verified all rules with SMT solvers before including them in the rule set. Alternatively, rewrite-rule frameworks such as Rare and IsaRare could be used for the specification and formal verification of SMT rewrite rules [25,17].

Table 1 lists a subset of rewrite rules—those most frequently applied on our benchmarks—along with their descriptions and sources of derivation. The complete list of rules is provided in Appendix Section A.1. Some entries correspond to multiple related transformations grouped under a single rule; in such cases,

Table 1: A subset of rewrite rules

| Name | Description | Source |
|---|---|---|
| ieee-NaN-2-3-5-6 | $(\Rightarrow u + \texttt{NaN} \longrightarrow \texttt{NaN}), (\Rightarrow \texttt{NaN} + u \longrightarrow \texttt{NaN}), (\Rightarrow u \times \texttt{NaN} \longrightarrow \texttt{NaN}), (\Rightarrow \texttt{NaN} \times u \longrightarrow \texttt{NaN})$ | I |
| ieee-infinity-19-20 | $(\Rightarrow -(+\infty) \longrightarrow -\infty), (\Rightarrow -(-\infty) \longrightarrow +\infty)$ | I |
| ieee-NaN-7 | $\Rightarrow -\texttt{NaN} \longrightarrow \texttt{NaN}$ | I |
| ieee-infinity-36-39-43-46 | $(u = 0 \Rightarrow u \times +\infty \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow +\infty \times u \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow u \times -\infty \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow -\infty \times u \longrightarrow \texttt{NaN})$ | I |
| simplify-7 | $(\Rightarrow u - (+0.0) \longrightarrow u), (\Rightarrow u - (-0.0) \longrightarrow u)$ | II |
| simplify-1 | $(\Rightarrow u + (-0.0) \longrightarrow u), (\Rightarrow u + (+0.0) \longrightarrow u)$ | II |
| simplify-1-2 | $(\Rightarrow -0.0 + u \longrightarrow u), (\Rightarrow +0.0 + u \longrightarrow u)$ | II |
| addSub-6 | $u_1 - (-0.0 - u_2) \longrightarrow u_1 + u_2$ | II |
| isNaN-add-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 + u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |
| isNaN-add-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 + u_2) = \texttt{NaN} \longrightarrow u_2 = \texttt{NaN}$ | III |
| isNaN-mul-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \Rightarrow (u_1 \times u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |
| isNaN-mul-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \Rightarrow (u_1 \times u_2) = \texttt{NaN} \longrightarrow u_2 = \texttt{NaN}$ | III |
| isZero-mul-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_2 < 0 \wedge (u_1 = +0.0 \vee u_1 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow -u_1$ | III |
| isZero-mul-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_1 < 0 \wedge (u_2 = +0.0 \vee u_2 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow -u_2$ | III |

each transformation is enclosed in parentheses and separated by commas. Each transformation is written as a logical implication: the expression to the left of $\Rightarrow$ specifies the preconditions that must hold for the transformation to apply, while the equation on the right defines the actual rewrite, where the left-hand side of the "$\longrightarrow$" is replaced by the right-hand side. The variables $u$, $u_i$, denote floating-point variables of type `Float32` or `Float64`, though one can generalize the rules to other precisions. In the rule syntax, the symbol "$\wedge$" denotes conjunction, "$\vee$" denotes disjunction, and "`RNE`" in the precondition indicates that the rounding mode must be round-to-nearest-even. When a precondition uses $\pm\infty$ or $\pm 0.0$ (e.g., $u \neq \pm\infty$), it abbreviates both signs: $u \neq \pm\infty$ means $(u \neq +\infty) \wedge (u \neq -\infty)$, and analogously $u \neq \pm 0.0$ means $(u \neq +0.0) \wedge (u \neq -0.0)$. For example, the rule `isNaN-add-1` states that if the floating-point variable $u_2$ is neither NaN nor infinity then the expression (`fp.isNaN` (`fp.add RNE` $u_1$ $u_2$)) can be replaced with (`fp.isNaN` $u_1$).

## 3 Benchmark Selection and Generation

Our empirical results (somewhat unsurprisingly) show that selecting relevant test inputs is important. Since most of our rewrite rules include special values in their preconditions or transformations, we need input files that exercise the floating-point theory including special values.

SMT-COMP is a common source of SMT benchmarks, however, this collection contains relatively few instances involving special values or explicit queries about them: out of 43,076 files from the non-incremental subset, only 5,067 mention any IEEE-754 special values, and only 62 of these allowed any rewrite rules to be applied. Furthermore, the solving times are so small that for cvc5 and Bitwuzla

not a single file has a non-trivial solving time. Therefore, we exclude SMT-COMP benchmarks from our main evaluation.

Instead, we focus on an existing real-world benchmark set, and attempt to generate additional input formulas from scratch from a grammar, as well as by mutating the existing files[5].

### 3.1 KeY-Derived Verification Conditions (KEY-FP)

Our first suite consists of verification conditions related to the floating-point theory generated by the KeY deductive verifier from realistic code [2]. We chose these instances because some of the checked properties focus on the absence or controlled use of special values, among other functional properties. We use the benchmarks as they were released.

### 3.2 Grammar-Generated Floating-Point Benchmarks (GGEN-FP)

We generated the second suite by extending the BanditFuzz SMT fuzzing framework [27] and creating grammar-based floating-point seed files with this extension. Concretely, we added some missing floating-point constructs, i.e., the special values `NaN`, positive and negative infinity, positive and negative zero, and the predicate `fp.isZero`. We also re-implemented a weight parameter that biases the generator toward particular constructs as a substitute for the default uniform random selection.

To obtain a higher fraction of formulas that involve or test for special values, we adjust the generation probabilities of floating-point constructs across three batches while keeping the structural parameter ranges fixed. In all batches, we reduce the probability of generating fused multiply–add (FMA) operations to 0.01 and eliminate the use of round-nearest-away (RNA) rounding mode, since MathSAT does not support them. We also double the probability of predicates related to special-value checks, namely `fp.isNaN`, `fp.isInfinite`, and `fp.isZero`. In the second batch, we build upon the first configuration by further doubling the generation probability of basic arithmetic operations (`fp.add`, `fp.mul`, `fp.sub`, and `fp.div`). In the third batch, we extend this setup by additionally doubling the probability of generating special floating-point values (`NaN` etc.)

We vary three structural parameters for the generated formulas: (i) the number of assertions, (ii) the depth of assertions, and (iii) the number of variables. For each parameter, we select the minimum value of 5 and maximum value of 15 and instantiate the Cartesian product over these ranges, and repeat the generation process five independent times. Since BanditFuzz introduces randomness, repeated runs result in different benchmarks. We randomly select the precision for each case to be either `Float32` or `Float64` uniformly.

During our empirical study, we observed that among the 16,877 generated files with at least one applied rewrite rule, many were trivial. Across 67,548

[5] All benchmark suites are publicly available at `https://github.com/rosaAbbasi/ginger/tree/main/benchmarks`.

corresponding solver runs, the overall average runtime was only 0.05 seconds, and just 72 instances exhibited an average runtime exceeding 10 seconds. That is, most generated formulas were too simple to meaningfully challenge the solvers. We thus only use the 72 instances with longer runtimes for actual solver testing.

### 3.3 Mutations of KEY-FP Instances (KEYMUT-FP)

Our third benchmark suite was motivated by the difficulty of generating realistic instances using the grammar-based approach. This suite consists of mutations of 12 selected test files from the KeY-FP benchmark suite. Before applying mutations, we manually removed uninterpreted sorts and all assertions related to heap and Java class structures, retaining only the pure floating-point components. This simplification ensures compatibility with Bitwuzla since it does not support the theory of integers and equalities over uninterpreted sorts.

To generate mutants, we used the seed-based mutation testing feature of Sparrow [34], setting the theory to `QF_FP` and configuring it to generate 100 mutants per seed file. Since the original implementation of Sparrow did not support saving all generated mutants, we modified the code to enable this functionality. The produced mutants were neither semantically nor syntactically preserving. We filtered out all mutants that did not conform to the SMT-LIBv2 grammar and caused solvers to produce syntax errors.

## 4 Implementation

Our prototype, Ginger, is implemented in Python and communicates with off-the-shelf SMT solvers by generating SMT-LIBv2 files and invoking their command-line interfaces. Ginger implements the entire workflow described in Section 2.Ginger applies the rewrite rules, illustrated in Table 1, to the SMT-LIB abstract syntax tree. The system is designed to be modular—new rules can be incorporated simply by extending the rule list. Ginger applies the rules sequentially in the order they appear and recursively traverses each `assert` statement to ensure that all applicable transformations are performed. For each benchmark instance, Ginger logs the set of applied rules for subsequent analysis.

*Recorded Metrics* Ginger records solver runtimes on the original benchmarks and then re-measures these runtimes after applying the normalization and rewriting. Runtimes are measured using the bash `time` command. To account for runtime variability, each solver–benchmark combination is executed five times, with a timeout of 300 seconds. Consequently, for each file and solver, we obtain two sets of five runtime measurements: one set before the rewriting and one set after. For each benchmark file and solver, we compute the relative difference for each of the five runs as $r_i = (t_{\texttt{before}} - t_{\texttt{after}})/t_{\texttt{before}}$ where $t_{\texttt{before}}$ and $t_{\texttt{after}}$ denote the runtimes before and after rewriting for run $i$. Positive values of $r_i$ indicate a speedup, whereas negative values indicate a slowdown. This yields five relative difference values per solver per file, capturing run-to-run variability in performance change.

In addition to solver runtimes, GINGER records the *normalization time* and *rewrite time* (in seconds) for each instance, measuring the overhead introduced by preprocessing and transformation. For each solver, we also log the result on both the original and rewritten files (i.e., `sat`, `unsat`, `unknown`, `timeout`, or `error`), as well as the set of rewrite rules applied to each instance.

## 5 Empirical Results

We test the SMT solvers cvc5, Z3, MathSAT, and Bitwuzla with our approach. Our focus is on the following research questions:

**RQ1:** *How does rewriting affect solver performance on real-world, grammar-generated, and mutation-based benchmarks?* This RQ captures the per-solver effects of rewriting with at most two rules across different benchmark suites.

**RQ2:** *How does rewriting affect the solver performance across the different solvers?* This RQ investigates cross-solver variability.

**RQ3:** *How does applying the full set of semantic-preserving rewrite rules on real-world and mutation-based test files affect performance?* This RQ investigates the potential of using (unrestricted) rewriting for performance optimization.

### 5.1 Experimental Setup

We run our experiments on a server with 256GB memory and 2x8 CPU cores at 3.3GHz. However, GINGER runs in a single thread and does not use more than 8GB of memory. We use four SMT solvers: Z3 [20] (version 4.15.2), cvc5 [6] (version 1.3.0), MathSAT [10] (version 5.6.11) and Bitwuzla [22] (built from latest GitHub commit [21]). We run all solvers using a single thread.

As described in the implementation section, we compute five relative difference values per solver per file, capturing run-to-run variability in performance change. These values serve as the input data for our statistical analysis, where we perform a two-sample t-test at a significance level of $\alpha = 0.05$ to determine whether the mean relative difference differs significantly from zero, under the null hypothesis that it is equal to zero (i.e. no statistically significant slowdown or speedup).

When test cases are too small or trivial, i.e. yield a runtime within $\epsilon = 10^{-2}$ of zero, they are excluded from analysis. Consequently, only SMT files with a non-zero baseline (before-rewriting) runtime are considered *testable* cases. Among these, we categorize benchmarks that exhibit statistically significant relative differences across all solvers as *uniformly significant benchmarks*. We classify those that show significance for only a subset of solvers as *partially significant*. The remaining benchmarks, for which the null hypothesis cannot be rejected by any solver, are referred to as *uniformly insignificant*. These three categories are mutually exclusive.

We consider the three benchmark sets KEY-FP, KEYMUT-FP and GGEN-FP described in Section 3. We do not run Bitwuzla on the KEY-FP set, since Bitwuzla does not support the theory of integers and equalities over uninterpreted

Table 2: Median and average runtimes per solver across benchmark suites

| Solver | KeY-FP | | KeYMut-FP | | GGen-FP | |
|---|---|---|---|---|---|---|
| | avg. | median | avg. | median | avg. | median |
| Z3 | 111.91 | 97.21 | 18.82 | 10.19 | 0.18 | 0.01 |
| cvc5 | 1.22 | 1.06 | 1.65 | 1.39 | 1.74 | 1.33 |
| MathSAT | 1.76 | 1.40 | 1.83 | 1.79 | 27.83 | 19.38 |
| Bitwuzla | n/a | n/a | 0.16 | 0.17 | 0.35 | 0.27 |

Table 3: Statistically significant changes in solving time after rewriting

| Benchmark Suite | #Test Files | #Uniformly Sig. | #Uniformly Insig. | #Partially Sig. | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Z3 | | cvc5 | | MathSAT | | Bitwuzla | |
| | | | | Sig. | Insig. | Sig. | Insig. | Sig. | Insig. | Sig. | Insig. |
| KeY-FP | 64 | 40 | 2 | 17 | 5 | 7 | 15 | 16 | 6 | N/A | N/A |
| GGen-FP | 72 | 3 | 31 | 0 | 33 | 27 | 6 | 26 | 7 | 18 | 15 |
| KeYMut-FP | 650 | 408 | 0 | 242 | 0 | 215 | 27 | 216 | 26 | 34 | 208 |

sorts that these files contain. For each solver, Table 2 reports the median and average runtime on original files (i.e., before rewriting), considering only those files to which at least one rule was applied. As shown, for each benchmark suite, there is at least one solver with noticeably longer runtimes, indicating that the test files are non-trivial and that solvers' runtimes vary substantially, supporting our decision to compare each solver against itself rather than against other solvers.

### 5.2 RQ1: Rewriting Effect on Different Benchmark Sets

To identify potential performance issues, we evaluate whether rewriting leads to statistically significant changes in solver performance, in either direction.

Table 3 summarizes the number of files for which rewriting shows a statistically significant difference in solving times. The "#Test Files" column shows the number of files to which at least one rewrite rule was applied; only these files are benchmarked. Table 3 shows that for the KeY-FP and KeYMut-FP sets, rewriting leads to a statistically significant change in solving time for at least one solver for all but a single KeY-FP file. Many of these changes are not uniform across solvers, i.e. different solvers see a significant change for a different number of files. For GGen-FP, the performance changes caused by rewriting are mostly statistically insignificant, pointing to the simplicity of the grammar-generated test files.

Figures 2 and 3 show the magnitude of the relative differences in solving time, for each solver sorted from the largest speedup (positive mean) to the greatest slowdown (negative mean), for the KeY-FP and KeYMut-FP sets. The plots only include testable files that had a significant difference for each individual solver (so the number of files per solver differs), and show the average over the five runs for each file.

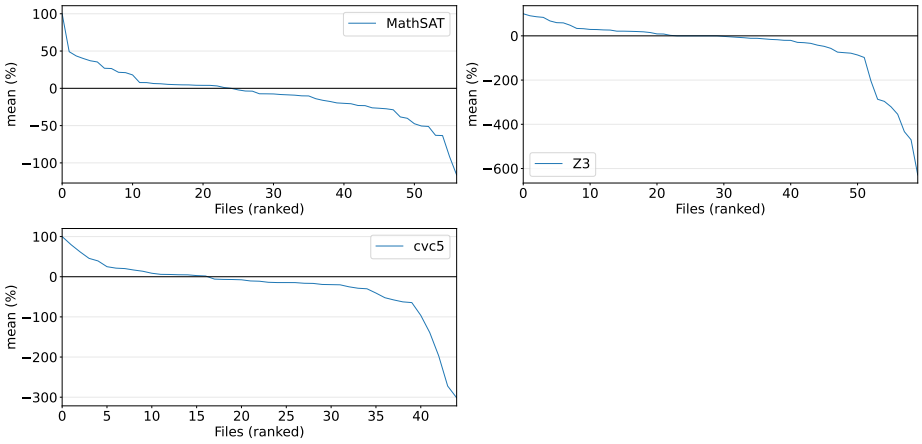



Fig. 2: Relative difference in solving time after rewriting, KeY-FP suite

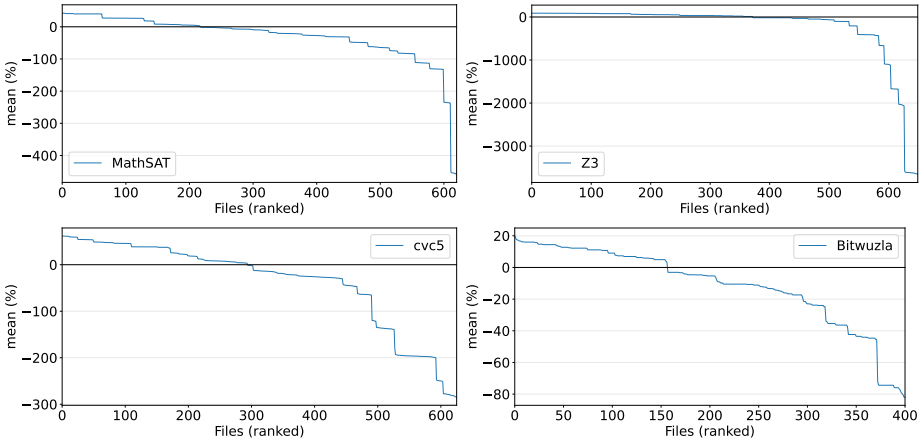



Fig. 3: Relative difference in solving time after rewriting, KeYMut-FP suite

The observed slowdowns for the KeY-FP and KeYMut-FP benchmark suites indicate potential performance issues. Despite the rewrite rules being designed as simplifications, in about half of the files they cause slowdowns, some of them of significant magnitude.

We essentially do not detect performance issues with the GGen-FP suite (see Appendix Section A.2 for the magnitude of the relative differences). For the few files that have a statistically significant relative change, rewriting causes speedups for all solvers—except for one file for Z3. This suggests that, in contrast to the real-world test files in KeY-FP and KeYMut-FP, grammar-generated test files are too simple for detecting performance issues and possibly also not representative of real-world SMT queries.

Moreover, we repeated our experiments (on KeY-FP and GGen-FP) using different random seeds for shuffling the rule list and reanalyzed the results. Qualitatively, the outcomes remained unchanged, indicating that our findings are robust with respect to the applied rules.

To isolate the effect of the rewrite rules, we additionally repeated the main experiment on the KeY-FP benchmark suite, but removed the extra assertions introduced during normalization from the rewritten files. We observed the same overall trend of speedups and slowdowns, showing that rewrites applied to existing assertions alone are sufficient to expose performance anomalies. We also evaluated the effect of normalization alone by running Ginger on all KeY-FP test files that originally had at least one rule applied. Normalization by itself produces both speedups and slowdowns in solver runtimes. The magnitudes of these changes are generally small, and fewer files exhibit significant differences. Since normalization is a semantics-preserving transformation that makes implicit information explicit or directly rewrites assertions, the slowdowns it causes also qualify as performance issues. The magnitudes of the relative differences for the rewrite-only and normalization-only experiments are reported in Appendix Sections A.3 and A.4, respectively.

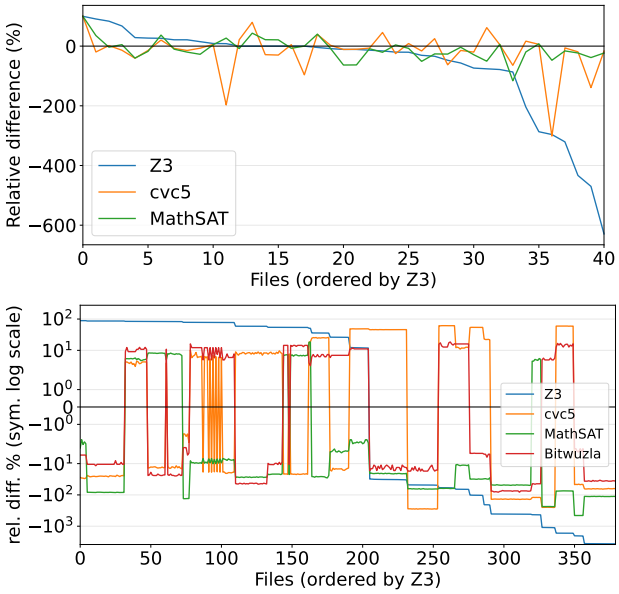
In the KeY-FP benchmarks, we also identified 7 cases for Z3 and one for MathSAT where solvers initially reported `unsat` but timed out after rewriting. In one instance, Z3 initially timed out but reported `sat` after rewriting. We saw no mismatched solver results in the other benchmark suites.

*Analysis of a Reported Performance Issue.* We submitted one representative performance issue to the developers of Z3, cvc5, and MathSAT. The Z3 issue received an automated response indicating a potential "performance bug". For cvc5, the reported performance issue [1] was also acknowledged. We further investiagted the slowdown ourselves. The slowdown persisted even after minimizing the queries and removing auxiliary normalization assertions. Analysis of cvc5's preprocessing and word-blasting phases revealed that the rewrite breaks sharing of a common subexpression across assertions, causing structurally different copies to be word-blasted separately. This cause was also confirmed by the cvc5 developers that reproduced the reported slowdown several weeks after paper submission. They also confirmed that the slowdown is no longer reproducible following a SymFPU update released after the submission of this paper. This analysis highlights the value of metamorphic testing: comparing semantically equivalent queries within a single solver can reveal and help explain performance anomalies that would be difficult to identify through cross-solver comparisons.

> **Answer to RQ1:** Rewriting is effective in triggering performance slowdowns and detecting performance issues, especially when applied to real-world test files and mutations derived from them.

### 5.3 RQ2: Cross-Solver Variability

We now evaluate whether or not rewriting leads to consistent speedups or slowdowns across different solvers. Our goal is to evaluate how variations in floating-




KEY-FP

KEYMUT-FP

Fig. 4: Comparison of relative differences in solving times across solvers

point theory implementations affect solvers' performance. For this, we examine test files that had a statistically significant difference across *all* solvers.

Figure 4 shows the *uniformly significant* files from the KEY-FP and KEYMUT-FP benchmark suites. The files are ordered according to the relative difference values for Z3. For improved clarity, for the KEYMUT-FP plot, we show the symmetric log-scale of relative differences, that preserves the signs. We excluded GGEN-FP here because it contains only three uniformly significant test files, which mostly cause speedups and do not provide interesting performance variation.

Figure 4 demonstrates the unpredictability and variability of floating-point implementations across solvers: For the same test file and after applying the same rewrite rules, not only does the magnitude of the relative performance change vary between solvers, but so does the direction of the change—some solvers solve the simplified file faster and some slower. In KEY-FP, for 23 files, there is a mix of slowdowns and speedups across solvers, and in the 380 uniformly significant KEYMUT-FP files, there are 280 files for which some solvers show a speedup while others show a slowdown. The step-like pattern in the KEYMUT-FP plots arises because many mutants generated from the same seed file differ only slightly and therefore have similar solving time before and after rewriting.

> **Answer to RQ2:** Rewriting affects each solver's performance differently, highlighting differences in their floating-point reasoning implementations.

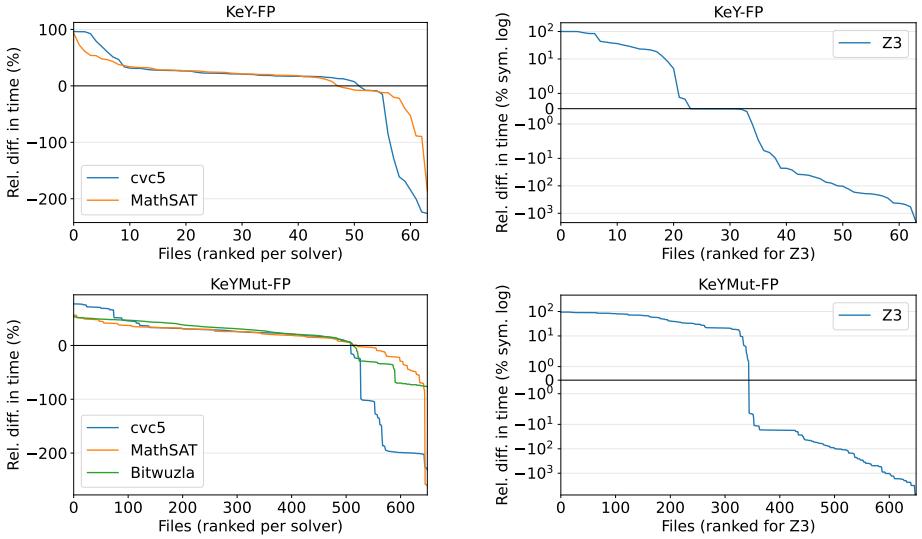



Fig. 5: Relative difference between overall time and original solving time

### 5.4 RQ3: Impact of Full Rewriting on Total Runtime

We now evaluate whether allowing the full list of rewrite rules—without the two-rule limit—can improve total solving time. We compare the *overall time*, i.e., the sum of runtimes for rewriting, normalization and solving the rewritten file, with the solving time of the original file. I.e., we investigate whether solvers benefit from the simplifications produced by unrestricted rule application. We note here that Ginger is a prototype implementation, and while designed carefully, it was *not* optimized for performance. Nevertheless, for KeYMut-FP, normalization and rewriting take only 0.032 s per file on average (0.034 s median).

We apply the complete set of rewrite rules—shuffled between per-file applications and without limiting the number of applications—to the KeY-FP and KeYMut-FP benchmark suites. For each solver, we compute the relative differences between the original solver runtime and the overall time (averaging solver runtimes, rewriting, and normalization times per file over five runs).

Figure 5 shows the relative differences for the KeY-FP suite at the top and KeYMut-FP suite at the bottom. The y-axis in the Z3 plots uses a symmetric log scale. We observe performance improvements for many input files. For the KeYMut-FP files that have improved overall solving times, the median relative differences are 48.62% (Z3), 15.13% (Bitwuzla), 20.74% (cvc5) and 25.94% (MathSAT). For KeY-FP files, the numbers are similar, and we show the full summary in Appendix Table 5.

> **Answer to RQ3:** Rewriting using rules such as ours may be a beneficial pre-processing step at least for the queries with floating-point special values.

## 6 Related-Work

SMT solvers have been subjected to extensive correctness and performance testing as well as proof checking [28,16]. Existing testing frameworks vary in the theories they support, the classes of bugs they target, and the techniques they employ. Several works explicitly address the floating-point theory [18,27,23,33,31,34,30]. Most focus on correctness, while a few consider performance [27,30]. More recently, Amrollahi et al. [5] identified solver stability as a major challenge, showing that semantically equivalent SMT formulas can exhibit substantially different runtimes due to minor syntactic variations.

BanditFuzz [27] targets performance slowdowns using reinforcement-learning-guided fuzzing and a reference solver for performance comparison. ET [30] applies grammar-based enumeration and differential testing. In contrast, our approach compares a solver against itself using semantics-preserving rewrite rules and does not rely on a reference solver or differential testing. Moreover, as we observed, grammar-generated inputs do not always yield realistic floating-point benchmarks.

Metamorphic testing has also been applied to floating-point SMT solving. Storm [18] generates new satisfiable formulas by recombining subformulas from satisfiable seeds, Sparrow [34] produces equi-satisfiable variants through oracle-guided mutations, and Bringolf et al. [7] use weakening and strengthening transformations. Unlike our work, these approaches primarily target correctness rather than performance anomalies.

Mikek et al. [19] simplify SMT formulas using compiler optimizations, while Pereira et al. [26] improve SMT performance through simplification, normalization, and caching. Our work differs in that it uses semantics-preserving rewrites to expose performance issues, although we also observe that such transformations can improve solver performance.

## 7 Conclusion

To conclude, our results demonstrate that metamorphic testing is an effective method for detecting performance issues in SMT solvers. We have submitted selected performance issues to the corresponding SMT solver developers, and at least one was acknowledged, investigated, and subsequently closed. While rewrite rules generally yield consistent speedups on syntactically regular, generated benchmarks, their impact on realistic benchmarks—and on mutants derived from them—is strongly solver- and benchmark-dependent. Our results also highlight the unpredictability of floating-point reasoning across solvers and the influence of benchmark structure: for some solvers, applying rewrite rules before solving is beneficial, whereas for others it may be more appropriate to incorporate such transformations directly into their internal heuristics.

**Disclosure of Interests.** The authors have no competing interests to declare.

## References


1. Unexpected Performance Slowdown on Semantically Equivalent SMT2 Files After (Simplification) Rewrites (issue 12383). https://github.com/issues/created?issue=cvc5%7Ccvc5%7C12383 (2026)
2. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive Verification of Floating-Point Java Programs in KeY. In: TACAS (2021). https://doi.org/10.1007/978-3-030-72013-1_13
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-Guided Synthesis. In: FMCAD (2013), https://ieeexplore.ieee.org/document/6679385/
5. Amrollahi, D., Preiner, M., Niemetz, A., Reynolds, A., Charikar, M., Tinelli, C., Barrett, C.: Towards SMT Solver Stability via Input Normalization. In: FMCAD (2025). https://doi.org/10.34727/2025/ISBN.978-3-85448-084-6_14
6. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: TACAS (2022). https://doi.org/10.1007/978-3-030-99524-9_24
7. Bringolf, M., Winterer, D., Su, Z.: Finding and Understanding Incompleteness Bugs in SMT Solvers. In: ASE (2022). https://doi.org/10.1145/3551349.3560435
8. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI (2008)
9. Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T.H., Zhou, Z.Q.: Metamorphic Testing: A Review of Challenges and Opportunities. ACM Comput. Surv. **51**(1) (2018). https://doi.org/10.1145/3143561
10. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: TACAS (2013). https://doi.org/10.1007/978-3-642-36742-7_7
11. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM (JACM) **52**(3) (2005). https://doi.org/10.1145/1066100.1066102
12. Dinda, P., Hetland, C.: Do Developers Understand IEEE Floating Point? In: IPDPS (2018). https://doi.org/10.1109/IPDPS.2018.00068
13. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: PLDI (2005). https://doi.org/10.1145/1065010.1065036
14. Hähnle, R., Huisman, M.: 24 Challenges in Deductive Software Verification. In: ARCADE@CADE (2017). https://doi.org/10.29007/J2CM
15. IEEE, C.S.: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)
16. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL (T)-based SMT solvers. In: FMCAD (2016). https://doi.org/10.1109/FMCAD.2016.7886666
17. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL. In: TACAS (2024). https://doi.org/10.1007/978-3-031-57246-3_17
18. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In: ESEC/FSE (2020). https://doi.org/10.1145/3368089.3409763

19. Mikek, B., Zhang, Q.: Speeding up SMT Solving via Compiler Optimization. In: ESEC/FSE (2023). https://doi.org/10.1145/3611643.3616357
20. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
21. Niemetz, A., Preiner, M.: Bitwuzla public repository - commit used for experiments. https://github.com/bitwuzla/bitwuzla/tree/aba05fa9383888a6e008533dd66196bc7fae11f2 (2025), accessed: 2025-10-14
22. Niemetz, A., Preiner, M.: Bitwuzla. In: CAV (2023). https://doi.org/10.1007/978-3-031-37703-7_1
23. Niemetz, A., Preiner, M., Barrett, C.: Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In: CAV (2022). https://doi.org/10.1007/978-3-031-13188-2_5
24. Nötzli, A., Brown, F.: LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In: SOAP@PLDI (2016). https://doi.org/10.1145/2931021.2931024
25. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language. In: FMCAD (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12
26. Pereira, J.M., Marques, F., Adão, P., Ait-El-Hara, H.R., Andrès, L., Carcano, A., Chambart, P., Maksimović, P., Santos, N., Santos, J.F.: Smt.ml: A Multi-Backend Frontend for SMT Solvers in OCaml. In: TACAS (2026). https://doi.org/10.1007/978-3-032-22752-2_2
27. Scott, J., Mora, F., Ganesh, V.: Banditfuzz: A Reinforcement-Learning based Performance Fuzzer for SMT Solvers. In: VSTTE (2020). https://doi.org/10.1007/978-3-030-63618-0_5
28. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: Smt proof checking using a logical framework. Formal Methods in System Design **42**, 91–118 (2013). https://doi.org/10.1007/S10703-012-0163-3
29. Sun, M., Yang, Y., Wang, Y., Wen, M., Jia, H., Zhou, Y.: SMT Solver Validation Empowered by Large Pre-Trained Language Models. In: ASE (2023). https://doi.org/10.1109/ASE56229.2023.00180
30. Winterer, D., Su, Z.: Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. Proc. ACM Program. Lang. (OOPSLA2) (2024). https://doi.org/10.1145/3689795
31. Winterer, D., Zhang, C., Su, Z.: On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. Proc. ACM Program. Lang. (OOPSLA) (2020). https://doi.org/10.1145/3428261
32. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L.: Fuzz4All: Universal Fuzzing with Large Language Models. In: ICSE (2024). https://doi.org/10.1145/3597503.3639121
33. Yao, P., Huang, H., Tang, W., Shi, Q., Wu, R., Zhang, C.: Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In: ISSTA (2021). https://doi.org/10.1145/3460319.3464803
34. Yao, P., Huang, H., Tang, W., Shi, Q., Wu, R., Zhang, C.: Skeletal Approximation Enumeration for SMT Solver Testing. In: ESEC/FSE (2021). https://doi.org/10.1145/3468264.3468540

# A Appendix

## A.1 All Rewrite Rules

Table 4 presents the complete list of rewrite rules, along with their descriptions and sources of derivation. Some entries correspond to multiple related transformations grouped under a single rule; in such cases, each transformation is enclosed in parentheses and separated by commas. Each transformation is written as a logical implication: the expression to the left of $\Rightarrow$ specifies the preconditions that must hold for the transformation to apply, while the equation on the right defines the actual rewrite, where the left-hand side of the "$\longrightarrow$" is replaced by the right-hand side. The variables $u$, $u_1$, $u_2$, and $u_3$ denote floating-point variables of type `Float32` or `Float64`, though one can generalize the rules to other precisions. In the rule syntax, the symbol "$\wedge$" denotes conjunction, "$\vee$" denotes disjunction, and "`RNE`" in the precondition indicates that the rounding mode must be round-to-nearest-even. When a precondition uses $\pm\infty$ or $\pm 0.0$ (e.g., $u \neq \pm\infty$), it abbreviates both signs: $u \neq \pm\infty$ means $(u \neq +\infty) \wedge (u \neq -\infty)$, and analogously $u \neq \pm 0.0$ means $(u \neq +0.0) \wedge (u \neq -0.0)$. For example, the rule `isNaN-add-1` states that if the floating-point variable $u_2$ is neither NaN nor infinity then the expression (`fp.isNaN` (`fp.add RNE` $u_1$ $u_2$)) can be replaced with (`fp.isNaN` $u_1$).

Table 4: List of rewrite rules

| Name | Description | Source |
|---|---|---|
| addSub-1 | $(u_1 = +0.0 \vee u_1 = -0.0) \wedge u_2 \neq (\texttt{NaN} \wedge \pm 0.0) \Rightarrow u_1 + u_2 \longrightarrow u_2$ | II |
| addSub-2 | $(u_2 = +0.0 \vee u_2 = -0.0) \wedge u_1 \neq (\texttt{NaN} \wedge \pm 0.0) \Rightarrow u_1 + u_2 \longrightarrow u_1$ | II |
| addSub-6 | $u_1 - (-0.0 - u_2) \longrightarrow u_1 + u_2$ | II |
| ieee-NaN-2-3-5-6 | $(\Rightarrow u + \texttt{NaN} \longrightarrow \texttt{NaN}), (\Rightarrow \texttt{NaN} + u \longrightarrow \texttt{NaN}), (\Rightarrow u \times \texttt{NaN} \longrightarrow \texttt{NaN}), (\Rightarrow \texttt{NaN} \times u \longrightarrow \texttt{NaN})$ | I |
| ieee-NaN-7 | $\Rightarrow -\texttt{NaN} \longrightarrow \texttt{NaN}$ | I |
| ieee-NaN-8 | $\Rightarrow \texttt{NaN}^{-1} \longrightarrow \texttt{NaN}$ | I |
| ieee-infinity-19-20 | $(\Rightarrow -(+\infty) \longrightarrow -\infty), (\Rightarrow -(-\infty) \longrightarrow +\infty)$ | I |
| ieee-infinity-21-22 | $(\Rightarrow +\infty^{-1} \longrightarrow +0.0), (\Rightarrow -\infty^{-1} \longrightarrow -0.0)$ | I |
| ieee-infinity-24-25 | $(u = -\infty \Rightarrow u + (+\infty) \longrightarrow \texttt{NaN}), (u = -\infty \Rightarrow +\infty + u \longrightarrow \texttt{NaN})$ | I |
| ieee-infinity-26-27 | $(u \neq -\infty \wedge u \neq \texttt{NaN} \Rightarrow +\infty + u \longrightarrow +\infty), (u \neq -\infty \wedge u \neq \texttt{NaN} \Rightarrow u + (+\infty) \longrightarrow +\infty)$ | I |
| ieee-infinity-31-32 | $(u \neq +\infty \wedge u \neq \texttt{NaN} \Rightarrow -\infty + u \longrightarrow -\infty), u \neq +\infty \wedge u \neq \texttt{NaN} \Rightarrow u + (-\infty) \longrightarrow -\infty$ | I |
| ieee-infinity-34-35-37-38 | $(u > 0 \wedge u \neq \texttt{NaN} \Rightarrow u \times +\infty \longrightarrow +\infty), (u < 0 \wedge u \neq \texttt{NaN} \Rightarrow u \times +\infty \longrightarrow -\infty), (u > 0 \wedge u \neq \texttt{NaN} \Rightarrow +\infty \times u \longrightarrow +\infty), (u < 0 \wedge u \neq \texttt{NaN} \Rightarrow +\infty \times u \longrightarrow -\infty)$ | I |
| ieee-infinity-36-39-43-46 | $(u = 0 \Rightarrow u \times +\infty \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow +\infty \times u \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow u \times -\infty \longrightarrow \texttt{NaN}), (u = 0 \Rightarrow -\infty \times u \longrightarrow \texttt{NaN})$ | I |
| ieee-infinity-41-42-44-45 | $(u > 0 \wedge u \neq \texttt{NaN} \Rightarrow u \times -\infty \longrightarrow -\infty), (u < 0 \wedge u \neq \texttt{NaN} \Rightarrow u \times -\infty \longrightarrow +\infty), (u > 0 \wedge u \neq \texttt{NaN} \Rightarrow -\infty \times u \longrightarrow -\infty), (u < 0 \wedge u \neq \texttt{NaN} \Rightarrow -\infty \times u \longrightarrow +\infty)$ | I |
| ieee-infinity-47 | $(\Rightarrow +0.0^{-1} \longrightarrow +\infty), (\Rightarrow -0.0^{-1} \longrightarrow -\infty)$ | I |
| isNaN-add-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 + u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |
| isNaN-add-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 + u_2) = \texttt{NaN} \longrightarrow u_2 = \texttt{NaN}$ | III |
| isNaN-div-1 | $u_2 \neq (\pm 0.0 \wedge \texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 / u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |

Table 4: List of rewrite rules (continued)

| Name | Description | Source |
|---|---|---|
| isNaN-div-2 | $u_1 \neq (\pm 0.0 \wedge \texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1/u_2) = \texttt{NaN} \longrightarrow u_2 = \texttt{NaN}$ | III |
| isNaN-mul-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \Rightarrow (u_1 \times u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |
| isNaN-mul-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \Rightarrow (u_1 \times u_2) = \texttt{NaN} \longrightarrow u_2 = \texttt{NaN}$ | III |
| isNaN-sub | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \Rightarrow (u_1 - u_2) = \texttt{NaN} \longrightarrow u_1 = \texttt{NaN}$ | III |
| isNaN-fma | $u_1 = \texttt{NaN} \vee u_2 = \texttt{NaN} \vee u_3 = \texttt{NaN} \Rightarrow \texttt{fma}(u_1, u_2, u_3) \longrightarrow \texttt{NaN}$ | III |
| mul-div-rem-1 | $\Rightarrow (-1.0) \times u \longrightarrow -u$ | II |
| mul-div-rem-1-2 | $\Rightarrow (u \times (-1.0) \longrightarrow -u$ | II |
| mul-div-rem-2 | $\texttt{RNE} \Rightarrow (-0.0 - u_1) \times (-0.0 - u_2) \longrightarrow u_1 \times u_2$ | II |
| mul-div-rem-3 | $-0.0 - (-0.0 \times u) \longrightarrow -0.0 \times u$ | II |
| simplify-1 | $(\Rightarrow u + (-0.0) \longrightarrow u), (\Rightarrow u + (+0.0) \longrightarrow u)$ | II |
| simplify-1-2 | $(\Rightarrow -0.0 + u \longrightarrow u), (\Rightarrow +0.0 + u \longrightarrow u)$ | II |
| simplify-4 | $(u_1 = +0.0 \vee u_1 = -0.0) \wedge u_2 \neq \pm\infty \wedge u_2 \neq \texttt{NaN} \Rightarrow u_2 + (u_1 - u_2) \longrightarrow +0.0$ | II |
| simplify-7 | $(\Rightarrow u - (+0.0) \longrightarrow u), (\Rightarrow u - (-0.0) \longrightarrow u)$ | II |
| simplify-10 | $\Rightarrow -0.0 - (-0.0 - u) \longrightarrow u$ | II |
| simplify-13 | $u \neq \texttt{NaN} \wedge u \neq \pm\infty \Rightarrow (u - u) \longrightarrow +0.0$ | II |
| simplify-14 | $\Rightarrow u \times 1.0 \longrightarrow u$ | II |
| simplify-14-2 | $\Rightarrow 1.0 \times u \longrightarrow u$ | II |
| simplify-18 | $u \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \Rightarrow u/u \longrightarrow 1.0$ | II |
| simplify-19 | $(u_1 = +0.0 \vee u_1 = -0.0) \wedge u_2 \neq (\texttt{NaN} \wedge \pm 0.0 \wedge \pm\infty) \Rightarrow (u_2/u_1 - u_2) \longrightarrow -1.0$ | II |
| true-false | $\Rightarrow (\texttt{fp.isNaN}\ (\_\ \texttt{NaN}\ [eb]\ [sb])) \longrightarrow \texttt{True}$ | III |
| isInfinite-div | $u_1 \neq \pm\infty \wedge (u_1 > 9.99999940e-01 \vee u_1 > 9.9999999999999989e-01) \Rightarrow (\texttt{fp.isInfinite}\ u_2/u_1) \longrightarrow (\texttt{fp.isInfinite}\ u_2)$ | III |
| isInfinite-add-1 | $u_1 \neq \pm\infty \wedge u_1 > 0 \wedge u_2 < 0 \Rightarrow u_2 + u_1 = \pm\infty \longrightarrow (\texttt{fp.isInfinite}\ u_2)$ | III |
| isInfinite-add-2 | $u_1 \neq \pm\infty \wedge u_1 < 0 \wedge u_2 > 0 \Rightarrow u_2 + u_1 = \pm\infty \longrightarrow (\texttt{fp.isInfinite}\ u_2)$ | III |
| isInfinite-sub-1 | $u_1 \neq \pm\infty \wedge u_1 > 0 \wedge u_2 > 0 \Rightarrow u_2 - u_1 = \pm\infty \longrightarrow (\texttt{fp.isInfinite}\ u_2)$ | III |
| isInfinite-sub-2 | $u_1 \neq \pm\infty \wedge u_1 < 0 \wedge u_2 < 0 \Rightarrow u_2 - u_1 = \pm\infty \longrightarrow (\texttt{fp.isInfinite}\ u_2)$ | III |
| isZero-mul-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_2 < 0 \wedge (u_1 = +0.0 \vee u_1 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow -u_1$ | III |
| isZero-mul-2 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_1 < 0 \wedge (u_2 = +0.0 \vee u_2 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow -u_2$ | III |
| isZero-mul-3 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_2 > 0 \wedge (u_1 = +0.0 \vee u_1 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow u_1$ | III |
| isZero-mul-4 | $u_1 \neq (\texttt{NaN} \wedge \pm\infty) \wedge u_1 > 0 \wedge (u_2 = +0.0 \vee u_2 = -0.0) \Rightarrow u_1 \times u_2 \longrightarrow u_2$ | III |
| isZero-div-1 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \wedge u_2 < 0 \wedge (u_1 = +0.0 \vee u_1 = -0.0) \Rightarrow u_1/u_2 \longrightarrow -u_1$ | III |
| isZero-div-2 | $u_2 \neq (\texttt{NaN} \wedge \pm\infty \wedge \pm 0.0) \wedge u_2 > 0 \wedge (u_1 = +0.0 \vee u_1 = -0.0) \Rightarrow u_1/u_2 \longrightarrow u_1$ | III |
| isNaN-sqrt-2 | $u \neq (\texttt{NaN} \wedge \pm\infty) \wedge (u = 0.0 \vee u > 0) \Rightarrow \texttt{sqrt}(u) \neq \texttt{NaN}$ | III |
| isNaN-sqrt-1 | $(u = 0 \vee u > 0) \Rightarrow \texttt{sqrt(u)} = \texttt{NaN} \longrightarrow u = \texttt{NaN}$ | III |

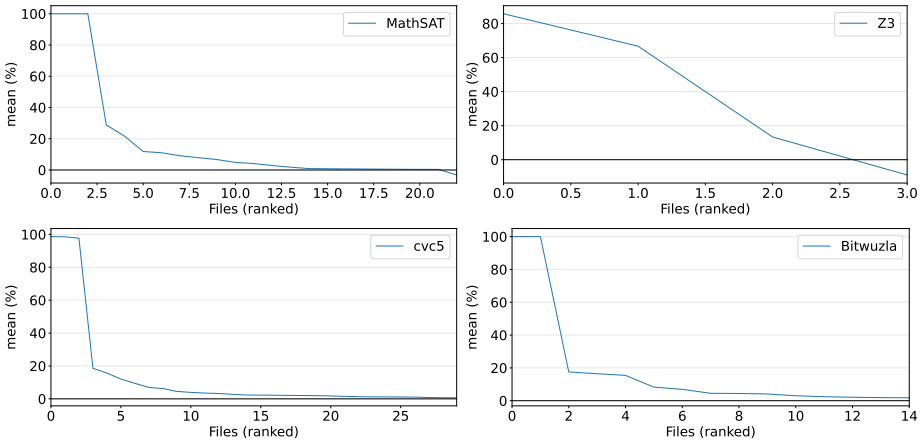



Fig. 6: Relative difference in solving time after rewriting, GGen-FP suite

### A.2 Relative Differences in Solving-Time for GGen-FP

Figure 6 shows the magnitude of the relative differences in solving time, for each solver sorted from the largest speedup (positive mean) to the greatest slowdown (negative mean), for the GGen-FP set. The plots only include testable files that had a significant difference for each individual solver (so the number of files per solver differs), and show the average over the five runs for each file.

### A.3 Effect of Rewrite Rules Independent of Normalization Assertions

Figure 7 shows the magnitude of the relative differences in solving time after removing the extra assertions introduced during normalization from the rewritten files, for each solver sorted from the largest speedup (positive mean) to the greatest slowdown (negative mean). The plots only include testable files that had a significant difference for each individual solver, and show the mean over the five runs for each file.

### A.4 Effect of Normalization on Solvers' Performance

Figure 8 shows the magnitude of the relative differences in solving time, for each solver sorted from the largest speedup (positive mean) to the greatest slowdown (negative mean). The plots only include testable files that had a significant difference for each individual solver, and show the mean over the five runs for each file. In comparison to the experiment where both rewrite rules and normalization were applied, the number of files with significant difference is much smaller, and the magnitude of the relative differences has also decreased substantially.

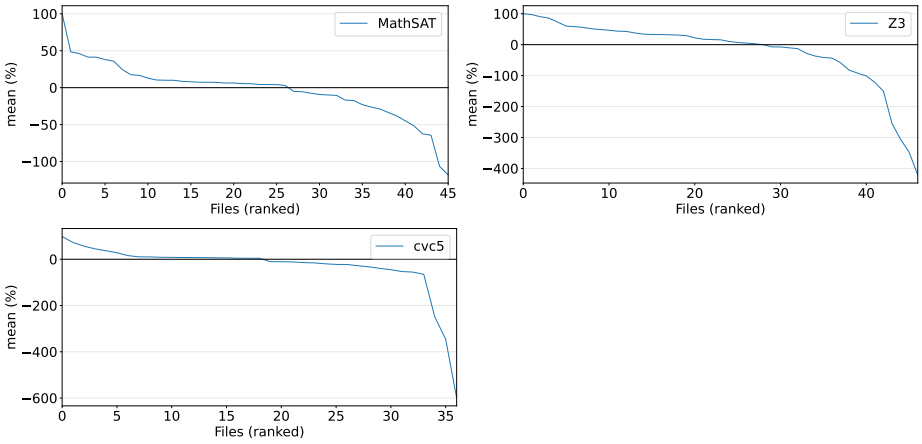



Fig. 7: Relative difference in solving time after removing extra assertions introduced during normalization, KEY-FP suite

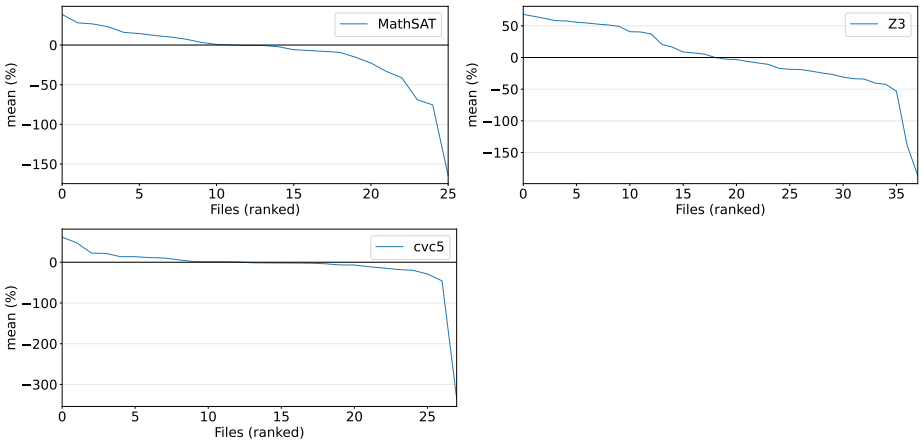



Fig. 8: Relative difference in solving time after only applying normalization, KEY-FP suite

### A.5 Impact of Full Rewriting on Total Runtime

Table 5 summarizes the positive relative differences, when applying the full set of rewrite rules to the KEY-FP and KEYMUT-FP benchmark suites.

Table 5: Summary of the positive relative differences, comparing original solving time against overall time

| Solver | KeY-FP | | KeYMut-FP | |
|---|---|---|---|---|
| | Mean. | Med. | Mean. | Med. |
| Z3 | 33.96 | 26.39 | 52.89 | 48.62 |
| cvc5 | 17.76 | 10.99 | 30.47 | 20.74 |
| MathSAT | 20.95 | 14.33 | 23.78 | 25.94 |
| Bitwuzla | N/A | N/A | 15.30 | 15.13 |