Dis. ETH No. 30617

SOLIDIFYING MODERN SMT SOLVERS

DOMINIK WINTERER

DISS. ETH NO. 30617

SOLIDIFYING MODERN SMT SOLVERS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

DOMINIK WINTERER

M.Sc., Albert Ludwigs University of Freiburg

born on 28 August 1991

accepted on the recommendation of

Prof. Dr. Zhendong Su (ETH Zurich)
Prof. Dr. Peter Müller (ETH Zurich)
Prof. Dr. Maria Christakis (TU Vienna)

2024

ABSTRACT

Satisfiability Modulo Theory (SMT) solvers realize one of the most powerful and mature classes of formal methods. They are foundational for many important advances in software research and industry. For example, one large-scale application is realized by AWS, which uses SMT solvers for verifying properties of cloud services. In almost all of their applications, SMT solvers are critical components solving NP-hard problems and establish trust through proof. Hence SMT solvers should be *correct* and *performant*, particularly in safety-critical and security-critical domains. Until recently, SMT solvers were widely trusted and believed to be solid.

This thesis puts the existing trust in SMT solvers to the test by proposing five approaches to address the correctness and performance of SMT solvers. The first approach is Semantic Fusion, a testing methodology to validate the correctness of SMT solvers. Semantic Fusion demonstrated that SMT solvers are less reliable than previously believed by finding dozens of soundness bugs in Z3 and CVC4. Orthogonal to Semantic Fusion, we propose Type-Aware Operator Mutation, a simple but unusually effective approach which found a total of 1,254 bugs in Z3 and CVC4 including soundness bugs across almost all theories in both solvers Z3 and CVC4/cvc5. To overcome the limitations of Type-Aware Operator Mutation, we propose Generative Type-Aware Mutation, an even more powerful approach, finding another 322 bugs in Z3 and CVC4/cvc5, among them several longstanding soundness bugs in CVC4. While the first three approaches focus on correctness, Janus is a pioneering approach for finding incompleteness bugs in SMT solvers. The final approach, Grammar-based Enumeration, addresses correctness and performance. Its realization ET is not only an effective bug finder but ET can help understand the evolution of SMT solvers. Our results suggest improved correctness in recent versions of Z3 and CVC4/cvc5 but decreased performance in newer Z3 releases.

This research enables *one of the world's largest academic bug-finding campaigns*. In five years, we found 1,825 unique bugs in Z3 and cvc5, among which 1,333 were fixed. Strikingly, we found 483 soundness bugs among which 349 were fixed. This thesis has significantly boosted research on solidifying formal methods beyond SMT solvers.

ZUSAMMENFASSUNG

Satisfiability Modulo Theory (SMT)-Solver gehören zu den mächtigsten und ausgereiftesten Klassen formaler Methoden. Sie sind grundlegend für viele wichtige Fortschritte in der Softwareforschung und -industrie. Ein großes Anwendungsbeispiel ist AWS, das SMT-Solver zur Überprüfung von Cloud-Dienst-Eigenschaften nutzt. In fast allen Anwendungen lösen SMT-Solver NP-schwere Probleme und schaffen Vertrauen durch Beweise. Daher sollten SMT-Solver *korrekt* und *leistungsfähig* sein, besonders in sicherheitskritischen Bereichen. Bis vor kurzem galten SMT-Solver als vertrauenswürdig.

Diese Arbeit testet das bestehende Vertrauen in SMT-Solver durch fünf Ansätze zur Verbesserung von Korrektheit und Leistung der Solver. Der erste Ansatz ist Semantic Fusion, eine Testmethode zur Validierung der Korrektheit von SMT-Solvern. Semantic Fusion zeigte, dass SMT-Solver weniger zuverlässig sind als angenommen, indem es Dutzende von Soundness-Bugs in Z3 und CVC4 fand. Orthogonal zu Semantic Fusion entwickeln wir Type-Aware Operator Mutation vor, eine einfache, aber effektive Methode, die insgesamt 1,254 Bugs in Z3 und CVC4 entdeckte, darunter Soundness-Bugs in fast allen Theorien. Zur Überwindung der Beschränkungen von Type-Aware Operator Mutation schlagen wir Generative Type-Aware Mutation vor, einen noch leistungsfähigeren Ansatz, der weitere 322 Bugs in Z3 und CVC4/cvc5 fand, darunter mehrere langjährige Soundness-Bugs in CVC4. Während sich die ersten drei Arbeiten auf Korrektheit konzentrieren, ist Janus ein innovativer Ansatz zum Finden von Unvollständigkeits-Bugs in SMT-Solvern. Der letzte Ansatz, Grammar-based Enumeration, befasst sich mit Korrektheit und Leistung. Das daraufbasierende Tool ET ist nicht nur ein effektiver Bug-Finder, sondern hilft auch, die Entwicklung von SMT-Solvern zu verstehen. Unsere Ergebnisse zeigen verbesserte Korrektheit in neueren Versionen von Z3 und CVC4/cvc5, aber eine verringerte Laufzeitperformanz in neueren Z3-Versionen.

Diese Forschung ermöglicht *eine der weltweit größten akademischen Bug-Findungs-Kampagnen*. In fünf Jahren fanden wir 1,825 Bugs in Z3 und cvc5, von denen 1,333 behoben wurden. Bemerkenswerterweise fanden wir 483 Soundness-Bugs, von denen 349 behoben wurden. Darüber hinaus hat diese Arbeit die Forschung zur Festigung formaler Methoden über SMT-Solver hinaus erheblich vorangetrieben.

PUBLICATIONS

This thesis is based on the following publications:

- **Dominik Winterer***, Chengyu Zhang*, Zhendong Su. "Validating SMT Solvers via Semantic Fusion". In: *ACM Programming Language Design and Implementation (PLDI), 2020.* [1]

  ★ ACM Distinguished Paper Award

  Invited to ACM TOPLAS Special Issue
- **Dominik Winterer***, Chengyu Zhang*, Zhendong Su. "On the Unusual Effectiveness of Type-Aware Mutations for Testing SMT Solvers" In: *ACM Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2020.* [2]

  ★ 1,000+ Bugs in the SMT Solvers Z3 and CVC4
- Jiwon Park*, **Dominik Winterer***, Chengyu Zhang, Zhendong Su. "Generative Type-Aware Mutation for Testing SMT Solvers" In: *ACM Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2021.* [3]

  ★ 250+ Bugs in the SMT Solvers Z3 and CVC4
- Mauro Bringolf, **Dominik Winterer**, Zhendong Su. "Finding and Understanding Incompleteness Bugs in SMT Solvers" In: *ACM/IEEE Automated Software Engineering (ASE), 2022.* [4]
- **Dominik Winterer**, Zhendong Su. "Grammar-based Enumeration of SMT Solvers for Correctness and Performance". In *ACM Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2024.* [5]

  ★ Bounded Validation of the SMT Solvers Z3 and cvc5

* equal contribution

The following publication was part of my doctoral research but presents results outside the scope of this thesis:

- Vince Szabó, **Dominik Winterer**, Zhendong Su. "CQ: A Metric for the Hardness of Programming Languages". In: *Arxiv, 2024.* [6]

ACKNOWLEDGEMENTS

I want to thank all the people who supported me during my Ph.D. studies.

First and foremost, I would like to thank my advisor Zhendong Su for his support and guidance. Insisting on the highest standards, persistence, and hard work, Zhendong has been a role model in so many areas. I appreciate his trust, help, and the opportunity to realize *Project YinYang for SMT Solver Testing*, the boldest possible research project that I could imagine.

I thank Peter Müller and Maria Christakis for accepting to serve as co-examiners of this thesis.

I had insightful discussions with Eric Bodden, Nikolaj Bjørner, Marcel Böhme, Michael Pradel, Benjamin Pierce, Nada Amin, André Platzer, Magnus Madsen, T. Y. Chen, Markus Püschel, and Ralf Jung. Thank you!

During my internship at AWS, I could apply SMT solver testing to a production setup. I am grateful to Robert Jones and Bruno Dutretre for this opportunity and enjoyed the discussions with Clark Barrett, Andy Reynolds, and Leo de Moura during that time. I am grateful to all SMT solver developers for their incredible work in addressing our bug reports.

A special thanks goes to my collaborators, especially Chengyu Zhang. From joyfully screaming in the ETH hallway when finding the first soundness bug in Z3, we have come a long way. I enjoyed collaborating with my former students Jiwon Park and Mauro Bringolf which resulted in two works contributing to this research. I thank the proofreaders: Hao Sun, Amar Shah, Tracy Ewen, Theo Sotiropoulos, Cong Li, and Gloria Esposito.

I thank my early mentors and great researchers from the AI Planning community, especially Robert Mattmüller, Martin Wehrle, Michael Katz, and Florian Pommerening. During my time at the University of Freiburg, they taught me rock-solid research fundamentals, opened doors, and treated me as an equal. Each of them has, in their own way, influenced my career and research style. To this day, I greatly benefit from having worked with them.

Big thanks to all current and former members of the AST lab, not yet mentioned: Shaohua Li, Manuel Rigger, Theodoros Theodoridis, Sverrir Thorgeirsson, Jiang Zhang, Michel Weber, Zu-Ming Jiang, Yann Giersberger and many others! I also want to thank the communication specialist Tracy Ewen and the assistants Christian Rossi and Ariane Nake.

I thank my friends Moritz Freidank, Dev and Deep Ganatra, Elias Rombach and the folks from *CityRunning West* for fresh perspectives from out-

side academia. I am grateful to my family especially to my father Michael X. Winterer for exposing me to computers at an early age.

I owe my gratitude to my partner Gloria Esposito who has been a rock in my life for over a decade. You have always been a great source of inspiration, counsel, help, and love.

Zurich, December 2024

Dominik Winterer

CONTENTS

1 INTRODUCTION

Formal methods, studied for decades, are experiencing a major comeback in computer science [7]. One of the most powerful and mature classes of formal methods are *Satisfiability Modulo Theory (SMT) solvers*. SMT solvers generalize SAT solvers and check the satisfiability of formulas with background theories relevant to programming languages and systems. SMT solvers are foundational for many important advances in software engineering, programming languages, and computer-aided verification. These include symbolic execution [8, 9], program synthesis. [10], solver-aided programming [11], program verification [12, 13], model checking [14], and neural network verification [15]. Recently, SMT solvers have also been increasingly adopted in industry. Applications include Zelkova, AWS' tool for verifying access policy properties [16], AdaCore Spark [17] for verifying cyber-physical systems, and Microsoft's SecGuru [18] for the verification of network policies. In all these applications, the SMT solver is a critical component solving an NP-hard problem and/or establishing trust through proof. However, buggy or slow SMT solvers can have severe consequences, particularly in the safety-critical and security-critical domains. Hence SMT solvers must be both *correct* and *performant*. However, ensuring their correctness and performance is challenging because of several reasons: (1) large codebases of the industrial-strength solvers Z3 and cvc5, (2) expressive input language of the solvers, and (3) the oracle problem: to validate the correctness of solver results.

Overarching goal of this thesis: *To solidify modern SMT solvers through designing principled, general automated testing techniques and tools.*

There are two key aspects in pursuing this goal: *correctness* and *performance*. This thesis contributes a research strand of five consecutive works, the first three on correctness, the fourth on performance, and the fifth on both. In all works, we view the SMT solvers as black box similar to real-world users. All proposed techniques belong to the family of automated software testers. They are general methodologies to facilitate adoption to other domains. Beyond its contributions to SMT solvers, we demonstrated that formal methods can benefit from automated software testing.

Approach	Focus	Oracle	Test generation	
Semantic Fusion	correctness	metamorphic	mutational	random
Type-Aware Operator Mutation	correctness	differential	mutational	random
Generative Type-Aware Mutation	correctness	differential	mutational generative	random
Weakening and Strengthening	performance	metamorphic	mutational	random
Grammar-based Enumeration	correctness performance	differential	generative	enumerative

Figure 1.1: The five approaches proposed in the thesis.

1.1 MAIN CONTRIBUTIONS

The main contributions of the thesis are realized in a single research strand of five consecutive works. We categorize them by their focus and how they address the test generation and oracle problems (Figure 1.1).

SEMANTIC FUSION (CHAPTER 3) *Semantic Fusion* is a novel, general testing methodology for validating SMT solvers. Semantic Fusion had a significant impact: it was the first approach to find a significant number (39) of soundness bugs in the solvers, mostly in nonlinear logic and string theory, mainly in Z3, and only few in CVC4.

TYPE-AWARE OPERATOR MUTATION (CHAPTER 4) Complementing Semantic Fusion, *Type-Aware Operator Mutation* is a testing technique with a differential oracle. Its key idea is mutating operators of the same type. It is simple but unusually effective: We found 1,254 bugs overall and 312 soundness bugs in almost every logic of Z3 and CVC4.

GENERATIVE TYPE-AWARE MUTATION (CHAPTER 5) Despite its effectiveness, Type-Aware Operator Mutation cannot grow or shrink SMT formulas. To overcome this, we devised *Generative Type-Aware Mutation* which can generate expressions with fresh operators and replace existing expressions. The technique found another 322 bugs in Z3 and CVC4/cvc5, among them several longstanding soundness bugs.

WEAKENING AND STRENGTHENING/JANUS (CHAPTER 6) While the previous works focus on correctness, this work is on performance. We define incompleteness bugs for SMT solvers and propose Janus, a tool

Category	Reported	Fixed
Both solvers	1,825	1,333
Z3	1,273	928
CVC4/cvc5	552	405
Default Mode (Z3)	750	569
Default Mode (CVC4/cvc5)	279	196
Soundness Bugs (Z3)	395	275
Soundness Bugs (CVC4/cvc5)	88	74
Performance Bugs (Z3)	74	25
Performance Bugs (CVC4/cvc5)	40	18

Figure 1.2: Overall statistics on bugs found in the fuzzing campaign enabled by the research presented in this thesis (July 2019 - July 2024).

for finding incompleteness bugs based on *Weakening and Strengthening*, a technique for mutating SMT solvers in a satisfiability-preserving manner. Janus is effective: it found 31 incompleteness bugs, of which 20 are already fixed.

Grammar-based Enumeration/ET (Chapter 7) The final piece addresses both correctness and performance. Different from the others, *Grammar-based Enumeration*'s testing is enumerative, not random. Our realization ET has many benefits: (1) It is a highly effective bug finder finding 102 bugs, out of which 76 were confirmed and 32 were fixed. (2) As ET enumerates from small to larger inputs, ET obsoletes bug reduction. (3) Finally ET can be used to understand the evolution of solvers. Our results suggest improved correctness in recent versions of both solvers but decreased performance in newer releases of Z3.

1.2 Impact

Bug findings The research presented in this thesis enabled *one of the world's largest academic fuzzing campaigns* with a significant impact on software research and formal methods. Since July 2019, *we found* 1,825 *bugs in the SMT solver Z3 and CVC4/cvc5* out of which 1,333 were fixed (see Figure 1.2). Perhaps most notably, we reported 483 soundness bugs out of which 286 were in the default modes of the solvers.

TOOLS & FRAMEWORKS We realized Semantic Fusion, Type-Aware Operator Mutation, and Generative Type-aware Mutation in YinYang, an open-source testing framework for SMT solvers. We realized Weakening and Strengthening in Janus which is based on YinYang.[1] At the time of this writing, YinYang and Janus[2] have *a combined 190 stars and 22 forks on GitHub*. Grammar-based enumeration is realized in ET and currently has 22 stars.

IMPACT IN THE RESEARCH COMMUNITY The research line described in this thesis opened up a new research (sub-)area for testing formal methods software and beyond. Works divide roughly into SMT testers [19, 20, 21, 22, 23, 24, 25], testers for other formal methods tools [26, 27, 28] and as benchmarks for bug finding works [19, 29]. Strikingly, SMT solvers such as cvc5 [30] and AltErgo [22] proposed dedicated fuzzers.

IMPACT IN INDUSTRY The research line described in this thesis had an impact on industry. At Google, developers working on validating SQL queries through SMT found YinYang useful and awarded it a *Google Open Source Peer Bonus*. Moreover, we received an Amazon Research Award for our proposal on "Practical Techniques for Reliable, Robust and Performant SMT Solvers". During my internship at AWS's automated reasoning group, I used ET for solidifying the string solver nfa2sat [31] which is part of Zelkova [16], and is used in production.

1.3 CHAPTER PREVIEWS

This section previews the five main chapters. For a smooth reading experience of the entire thesis, we recommend starting with Background (Section 2). Each of the five main chapters can then be read independently.

1.3.1 *Semantic Fusion*

MOTIVATION As of early 2019, there was little reason to doubt the reliability of SMT solvers. Z3 and CVC4, the two most popular and widely used solvers, had been developed for over a decade and only exhibited very few soundness issues. As a result, the solvers were regarded as mature and were widely trusted. This work is the first to challenge this trust in the solvers. Its results were surprising and had a significant impact.

[1] https://github.com/testsmt/yinyang
[2] https://github.com/testsmt/janus

$$\begin{aligned}
&\varphi_1 = x > 0 \land x > 1\\
&\varphi_2 = y < 0 \land y < 1\\
&\varphi_{concat} = (x > 0 \land x > 1) \land (y < 0 \land y < 1)\\
&\varphi_{fused} = (x > 0 \land z - y > 1) \land (z - x < 0 \land y < 1)
\end{aligned}$$

FIGURE 1.3: Semantic Fusionon two satisfiable formulas φ_1 and φ_2. Variable z realizes the fusion function $z = x + y$. Shaded: Randomly chosen occurrences of x and y to be replaced by variable inversion terms.

APPROACH We introduce Semantic Fusion, a general, effective approach to validating SMT solvers. Our key insight is to fuse two tests into a new test that combines the structures of its ancestors. We fuse two equisatisfiable formulas φ_1 and φ_2 (*i.e.*, both φ_1 and φ_2 are either satisfiable or unsatisfiable) into an equisatisfiable formula φ_{fused}. Semantic Fusion consists of the following three main steps:

1. *Formula Concatenation*: Concatenate φ_1 and φ_2 by formula conjunction or disjunction;

2. *Variable Fusion*: Create fresh variables to connect the variable sets of φ_1 and φ_2 using *fusion functions*; and

3. *Variable Inversion*: Substitute some occurrences of the chosen variables in φ_1 and φ_2 by *inversion functions*.

Figure 1.3 illustrates Semantic Fusion on two satisfiable formulas φ_1 and φ_2. We first concatenate φ_1 and φ_2, and obtain φ_{concat} as a result. Then, we introduce a fresh variable z and a *fusion function* $f(x, y) = x + y$, and construct a relation $z = f(x, y)$, which induces two equations $x = z - y$ and $y = z - x$. From these two equations, we obtain two *inversion functions* $r_x(y, z) = z - y$ and $r_y(x, z) = z - x$. Next, we replace the highlighted occurrences of x and y by the corresponding inversion functions $r_x(y, z)$ and $r_y(x, z)$, which results in formula φ_{fused}. By construction, the formula φ_{fused} is also satisfiable. We feed φ_{fused} to the SMT solver under test and observe the result. If the result is `unsat`, we have detected a (soundness) bug in the SMT solver under test.

MAIN RESULTS We introduced Semantic Fusion, a novel, general methodology for stress-testing SMT solvers. Based on the Semantic Fusion methodology, we design and develop the first highly effective framework, YinYang for SMT solver validation — the tool is customizable and conveniently supports various SMT theories. We conduct a testing campaign of Z3 and CVC4 using YinYang to demonstrate its effectiveness. *We found and reported 76 bugs in Z3 and CVC4 out of which 62 bugs were confirmed and 57 were fixed. All bugs were in the default arithmetic and string solvers. This work realizes the largest and most successful testing campaign against SMT solvers.*[3]

1.3.2 *Type-Aware Operator Mutation*

MOTIVATION Semantic Fusion [1] and STORM [32] have demonstrated that SMT solvers are clearly less reliable than previously presumed. Yet it is unclear whether SMT solvers have reached a strong level of maturity or whether many critical bugs remain. This chapter proposes a simple but unusually effective approach for testing SMT solvers finding 1,254 unique bugs in the state-of-the-art SMT solvers Z3 and CVC4.

APPROACH To answer this question, we introduce Type-Aware Operator Mutation, a simple, yet unusually effective approach for stress-testing SMT solvers. Its key idea is to mutate functions within SMT formulas with functions of conforming types. Figure 1.4 illustrates type-aware operator mutation on an example formula. We replace the "`distinct`" in φ by an operator of conforming type, *e.g.*, the equals operator "=" to obtain formula φ_{test}. We then differentially test SMT solvers with φ_{test} as input and observe their results. If the results differ, *e.g.*, one SMT solver returns `sat` while the other returns `unsat`, we found a soundness bug in either of the tested solvers. Formula φ_{test} is clearly unsatisfiable, as b cannot exist whenever a is odd. In fact, while CVC4 correctly returns `unsat` on φ_{test}, Z3 incorrectly reports `sat` on φ_{test}. Thus, φ_{test} has triggered a soundness bug in Z3 which was promptly fixed by Z3's main developer.

MAIN RESULTS We introduced Type-Aware Operator Mutation, a simple, but unusually effective approach for stress-testing SMT solvers. We realized Type-Aware Operator Mutation in OpFuzz which helps SMT solver developers and practitioners to stress-test SMT solver decision procedures. *We stress-tested Z3 and CVC4 using OpFuzz, and reported 1,254 bugs. Out of these, 963 bugs were confirmed, and 917 bugs were fixed. Perhaps most notably, we found soundness bugs in almost all logics of Z3 and many in CVC4. OpFuzz is*

[3] At the time of publication of this work at PLDI '20.

```
; \phi
(assert (forall ((a Int))
        (exists ((b Int))
        ( distinct  (* 2 b) a))))
(check-sat)
```

```
; \phi_{test}
(assert (forall ((a Int))
        (exists ((b Int))
        ( =  (* 2 b) a))))
(check-sat)
```

FIGURE 1.4: Type-aware operator mutation illustrated. We mutate the distinct operator in φ to the equals operator (see φ_{test}). Formula φ_{test} triggers a soundness bug in Z3 which reports `sat` on this unsatisfiable formula.

https://github.com/Z3Prover/z3/issues/3973

also the first to find a significant number of soundness bugs in CVC4, which has resisted several fuzzing campaigns up to this point.

1.3.3 *Generative Type-Aware Mutation*

MOTIVATION Several ongoing fuzzing campaigns have been continuously fuzzing SMT solvers Z3 and CVC4. This has led to many fixes in them. One such approach is OpFuzz which found several hundreds of bugs. However, despite its effectiveness, OpFuzz has several limitations. First, OpFuzz has a finite mutation space: the seed formulas have a fixed set of operators with only 2-3 choices for mutating each of them. Second, as a consequence of the bug fixes in the SMT solvers Z3 and CVC4, fuzzers are finding progressively fewer critical bugs. We propose Generative Type-Aware Mutation, to overcome these shortcomings.

APPROACH The key idea of Generative Type-Aware Mutation is mutating expressions in the AST of an SMT-LIB script by newly generated expressions of the same type. Let φ be a seed formula (see Figure 1.5).

STEP 1 CHOOSE A RANDOM EXPRESSION: We first choose a random expression $expr_1$ from the set of φ's expressions $expr(\varphi)$. Say we have picked the $expr_1 = \mathtt{x}$. The expression is of type `String` and will serve as the replacee for the newly generated expression.

STEP 2 CHOOSE A RANDOM OPERATOR: Next, we choose a suitable random operator. Such an operator should have the return type `String` and for all of its arguments, there should be at least one expression of conforming type in $expr(\varphi)$. Since φ contains terms of type `Bool`, `Int`, and `String`, the operator's arguments should be one of those types.

```
(declare-fun x () String)
(assert (> (- (str.to_int
    (str.++ x x))) 0))
(check-sat)
```

$expr_1 \in \{$ `x`, `0`, `(str.++ x x)`, $\cdots\}$

(a) Choose random expression

$op \in \{$ `str.from_int`, `str.++`, $\cdots\}$

```
(str.from_int Int String)
```

`int` $\in \{$ `0`, `(str.to_int (str.++ x x))`,
`(- (str.to_int (str.++ x x)))`$\}$

(b) Choose operator & integer expression

$$expr_2 = (\overset{op}{\texttt{str.from\_int}}\ \overset{\texttt{int}}{\texttt{0}})$$

(c) Generate new expression

```
(declare-fun x () String)
(assert (> (- (str.to_int
    (str.++ (str.from_int 0) x))) 0))
(check-sat)
```

(d) Mutant formula φ_{mutant} (Z3#5108)

FIGURE 1.5: Generative Type-Aware Mutation illustrated.

Candidates are the string to integer conversion function `str.from_int`, the string concatenation `str.++`, and all other operators taking `Bool`, `String` as arguments and returning `Bool`. Assume we have chosen the operator `str.from_int`.

STEP 3 GENERATE NEW EXPRESSION: Then, we generate an expression $expr_2$ with respect to the signature of the chosen operator. The signature for the operator `str.from_int` is defined as

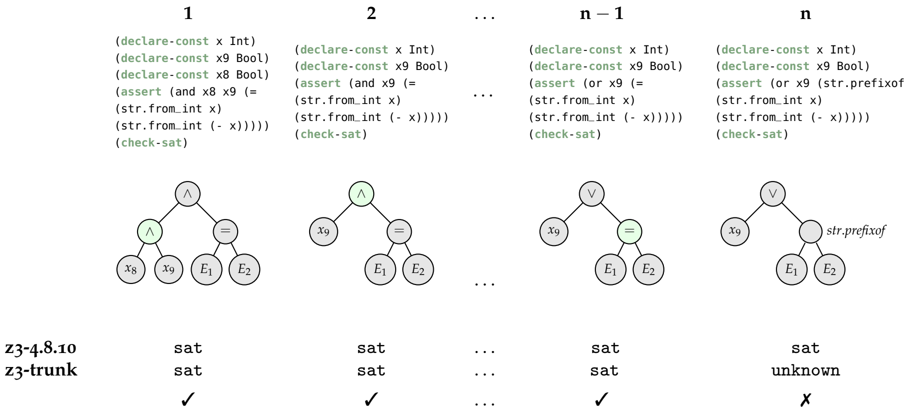
```
(str.from_int Int String)
```

Hence, we select an `Int` expression from $expr(\varphi)$. For the single parameter of type `Int`, we choose `0`. Then, with the chosen operator and expression, we construct the following new expression:

$$expr_2 = \texttt{(str.from\_int 0)}$$

STEP 4 SUBSTITUTION: Finally, we substitute $expr_1$ by $expr_2$ in φ which results in the formula φ_{mutant}. We feed φ_{mutant} to two or more SMT solvers and compare their results.

MAIN RESULTS We introduced *Generative Type-Aware Mutation*, a novel, effective approach for stress-testing SMT solvers. Generative Type-Aware Mutation is a hybrid of mutation-based and grammar-based fuzzing and has an infinite mutation space, overcoming one of OpFuzz's key limitations.



1

```
(declare-const x Int)
(declare-const x9 Bool)
(declare-const x8 Bool)
(assert (and x8 x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

2

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (and x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

...

n − 1

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (or x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

n

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (or x9 (str.prefixof
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

	1	2	...	n − 1	n
z3-4.8.10	sat	sat	...	sat	sat
z3-trunk	sat	sat	...	sat	unknown
	✓	✓	...	✓	✗

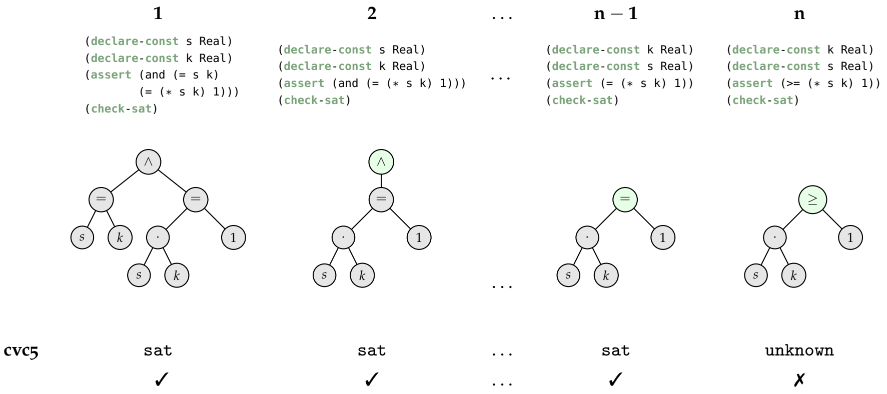
Figure 1.6: Finding regression incompleteness with Janus: sample mutation chain leading to a regression incompleteness in Z3 (Z3#5381).

Based on Generative Type-Aware Mutation, we have realized TypeFuzz. We reported 322 bugs in the Z3 and CVC4 with TypeFuzz. Among these, 290 bugs were confirmed and 278 bugs were fixed. *Most notably, Generative Type-Aware Mutation found 20 soundness bugs in CVC4's default mode alone. Several of them (7/20) are 2 years latent and predate all previous fuzzing campaigns.*

1.3.4 *Weakening and Strengthening/Janus*

MOTIVATION An SMT solver returns `unknown` on a formula if it cannot determine the satisfiability formula. Incompleteness bugs, *i.e.* unexpected `unknown`-results, impact the performance of SMT solvers' client applications frustrating their developers. In this work, we (1) formalize two types of incompleteness bugs in SMT solvers, and (2) propose a technique and tool for finding incompleteness bugs.

APPROACH This section presents Janus, our approach to tackling regression and implication incompletenesses. Regression incompletenesses are caused by (recent) code changes leading to an incompleteness on previously solved formulas. Typically such bugs affect client software that works correctly with an older version of the SMT solver but fails after updating the solver. Implication incompletenesses occur when an SMT solver solves a given input formula φ but minor changes in the formula (*i.e.* the mutation to



```
(declare-const s Real)
(declare-const k Real)
(assert (and (= s k)
        (= (* s k) 1)))
(check-sat)
```

```
(declare-const s Real)
(declare-const k Real)
(assert (and (= (* s k) 1)))
(check-sat)
```

```
(declare-const k Real)
(declare-const s Real)
(assert (= (* s k) 1))
(check-sat)
```

```
(declare-const k Real)
(declare-const s Real)
(assert (>= (* s k) 1))
(check-sat)
```

Figure 1.7: Finding an implication incompleteness with Janus: sample mutation chain leading to an implication incompleteness in cvc5 (cvc5#10891).

φ') cause the solver to report `unknown`. Such formula pairs suggest possible improvements for SMT solvers, *e.g.*, to formula rewriters, pre-processors, theory solvers *etc.* If φ was generated by a client application of an SMT solver, fixing such bugs makes the application more robust.

Janus has two concurrent modes, one for each bug type. We describe them in two separate examples. Finding regression incompletenesses Figure 1.6 shows a mutation chain of Janus for finding regression incompletenesses (from left to right). Janus starts with a seed formula on which both Z3 and legacy z3-4.8.10 return sat (step 1). Janus then chooses a *random* rule from its rule set and applies it to the seed (step 2). The process continues up to the point where Z3 returns `unknown` and z3-4.8.10 returns `sat`. Janus detected a regression incompleteness. This is a real case, *i.e.* an actual bug that we reported to the issue tracker of Z3. Figure 1.7 shows a mutation chain of Janus for finding regression incompletenesses (from left to right). Janus starts with a satisfiable seed formula (step 1). Janus then chose a satisfiability-preserving transformation rule, *e.g.*, dropping the first conjunct of the `and` expression. As the oracle is `sat`, we weaken, if the oracle was `unsat`, we would strengthen. This results in a mutated formula (step 2) satisfiable by construction. Janus generates mutants this way until the solver returns `unknown`. SMT solver developers can investigate the unknown case together with the rule (c.f. step $n - 1$ to n) that led to the `unknown`-result to understand why the SMT solver has failed. This is a real case. We reported it to the issue tracker of cvc5.

MAIN RESULTS We proposed Janus, an approach for finding incompleteness bugs in SMT solvers. *We stress-tested the two state-of-the-art SMT solvers Z3 and CVC5 with* Janus *and totally reported 31 incompleteness bugs. Out of these, 26 have been confirmed as unique bugs and 20 are already fixed by the developers.* Our diverse bug findings uncovered functional, regression, and performance bugs—several triggered discussions among the developers sharing their in-depth analysis.

1.3.5 *Grammar-based Enumeration/*ET

MOTIVATION SMT solvers must be both *correct* and *performant*, particularly in safety-critical and security-critical domains. In the last several years, there has been much effort in improving SMT solvers, especially through fuzzing [1, 2, 21, 32]. Z3 and cvc5 are the two most powerful SMT solvers and are very reliable. Developers of Z3 and cvc5 fixed hundreds of correctness and performance bugs found by fuzzers. As a result of these and other fixes, SMT solvers have greatly matured. However, despite this, all existing fuzzers are unsystematic focusing on random testing. Unsystematic testing can lead to missed bugs and does not provide any guarantees.

APPROACH This work changes the perspective on testing SMT solvers advocating for systematic, grammar-based enumeration rather than random-based testing. We propose ET, a grammar-based enumeration tool for SMT solvers. We compile context-free grammars of the SMT theories to algebraic datatypes and leverage FEAT [33], an approach for functional enumeration. This realizes a test generator which ET couples with an oracle to perform differential testing between the solvers. Given a context-free grammar G, a number of tests N, and two or more SMT solvers, ET stress-tests each solver with the *N-smallest* inputs *w.r.t.* grammar G. This approach has multiple unique benefits: (1) it exploits the *small scope hypothesis* which states that most bugs trigger on small-sized inputs [34, 35], (2) because of the small-sized bug triggers it is particularly suitable for identifying performance issues, and (3) it provides bounded correctness assurances. Using ET, we conducted a large-scale fuzzing campaign for correctness and performance bugs in the state-of-the-art SMT solvers Z3 and cvc5. We reported 102 bugs among which 76 bugs were confirmed and 32 bugs were already fixed. We found bugs in various SMT theories, including arrays, floating points, real and integer arithmetic, strings *etc.* Even though SMT solvers have been

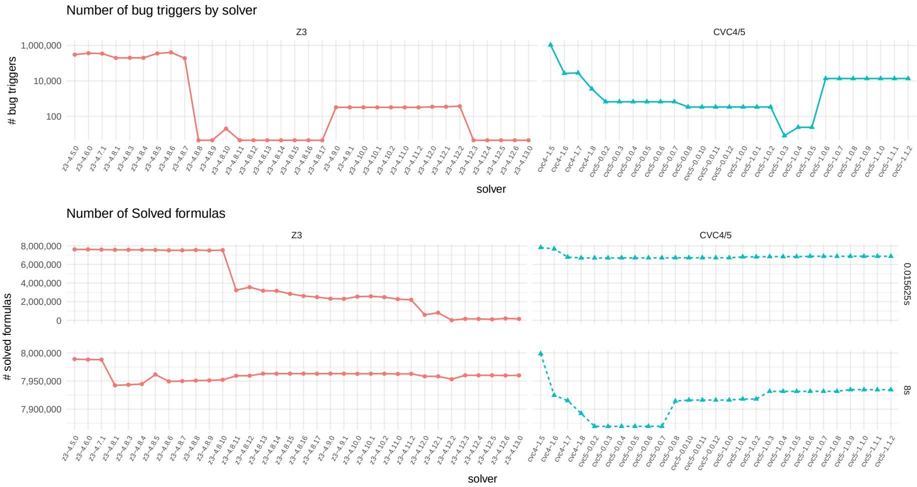


Figure 1.8: Evolution results for Z3 & CVC4/cvc5 releases from the last six years. Top: correctness in number of bug triggers. Bottom: performance in number of solved formulas.

extensively and continuously tested, we are still able to quickly find these bugs while the benefits of the other fuzzers seem to have saturated.

Quantifying solver evolution helps developers understand long-term effects and users to judge particular features. With ET, we tested all consecutive versions of the SMT solvers Z3 and CVC4/cvc5 from the last six years (61 solvers). We devised eight grammars for the official SMT theories and generated one million formulas per grammar.

main results We introduce ET, a grammar-based enumerator for validating SMT solver correctness and performance. ET is highly effective at bug finding and has many complimentary benefits. Despite the extensive and continuous testing of the state-of-the-art SMT solvers Z3 and cvc5, ET found 102 bugs, out of which 76 were confirmed and 32 were fixed. Moreover, ET can be used to understand the evolution of solvers. We derive eight grammars realizing all major SMT theories including the booleans, integers, reals, realints, bit-vectors, arrays, floating points, and strings. Using ET, we test all consecutive releases of the SMT solvers Z3 and CVC4/cvc5 from the last six years (61 versions) on 8 million formulas, and 488 million solver calls. *Our results suggest improved correctness in recent versions of both* *solvers but decreased performance in newer releases of Z3 on small timeouts (since z3-4.8.11) and regressions in early cvc5 releases on larger timeouts (see Figure 1.8).*

2 BACKGROUND

Before diving into the contributions of this thesis, we present background on SMT and automated testing. We refer unfamiliar readers to Kroening and Strichman's book for an excellent in-depth introduction to SMT [36].

2.1 SATISFIABILITY MODULO THEORIES

This subsection gives the necessary background on SMT and SMT solvers. We assume basic familiarity with propositional and first-order logic.

Definition 2.1.1 (Satisfiability modulo theories)**.** Satisfiability Modulo Theories (SMT) is the problem of determining the satisfiability of a formula φ in a first-order theory T. We call φ an SMT formula.

The theories T are aimed at modeling software and systems. They include the booleans, linear and nonlinear integer/real arithmetic, bitvectors, arrays, uninterpreted functions, floating points, strings, and can be combined.[1] As SMT generalizes SAT, it is NP-hard and can be undecidable for some theories and logics. For an SMT formula φ, we denote its free variables by $vars(\varphi)$. A substitution of a variable x in $vars(\varphi)$ by an expression e is denoted by $\varphi[e/x]$. A model M for φ is a function that maps all free variables $x_1, \cdots x_n \in vars(\varphi)$ to values in their domains such that $\varphi[M(x_1)/x_1, \cdots, M(x_n)/x_n]$ simplifies to *true*. If formula φ has at least one model, we call it *satisfiable* and *unsatisfiable* otherwise. A formula without free variables is called a sentence. As an example consider the formula:

$$\varphi = \forall a \in \mathbb{Z}\ \exists b \in \mathbb{Z} \quad 2 \cdot b = a$$

The formula says that for every integer `a`, there is an integer `b` such that `a` is twice as big as `b`. This means intuitively "*every integer is even*". As this is not the case, there is no model for φ and hence φ is unsatisfiable.

Definition 2.1.2 (SMT solver)**.** An SMT solver S is a tool for solving an SMT formula φ automatically. We distinguish the following cases:

[1] Theory T can be a combination of several sub-theories. This is realized through lazy SMT, an approach where decision procedures of separate theories share equalities [37]

- $S(\varphi) = \texttt{sat}$ if φ is satisfiable
- $S(\varphi) = \texttt{unsat}$ if φ is unsatisfiable
- $S(\varphi) = \texttt{unknown}$ if S cannot determine φ's satisfiability.

Prominent SMT solvers include Z3 [38], CVC4/cvc5 [39, 40], Yices [41], MathSat [42], OpenSMT [43], SMTInterpol [44], AltErgo [45], Boolector [30], and STP [46]. Among the solvers, Z3 and CVC4/cvc5 are the most powerful and widely used SMT solvers. Different from all others, they support all of SMT's theories while the others specialize in a few theories and logics. Hence, Z3 and CVC4/cvc5 are complicated pieces having 505,829 and 627,830 lines of mostly C++ code, respectively.

2.1.1 *SMT-LIB Initiative, Language, and Benchmarks*

SMT-LIB is an initiative to standardize input language and benchmarks for SMT solvers. The SMT-LIB language is an expressive LISP dialect for representing SMT formulas. Its most common commands include: "`declare-const`" to declare a variable, "`assert`" to realize the formula, and "`check-sat`" to query the satisfiability of asserted constraints, "`get-model`" to retrieve the model for satisfiable formulas and return an error otherwise. Consider the following examples to see how SMT formulas are realized:

$$\varphi = \forall a \in \mathbb{Z}\ \exists b \in \mathbb{Z} \quad 2 \cdot b = a \quad \rightsquigarrow$$

```
(assert (forall ((a Int))
        (exists ((b Int))
        (= (* 2 b) a))))
(check-sat)
```

SMT-LIB has many features including let bindings, algebraic data types, *etc.* We restrict ourselves to the aforementioned, basic subset of SMT-LIB. For more details including precise formal background, syntax, and semantics, we refer to the SMT-LIB standard v2.6 [47]. SMT-LIB also provides a large-scale benchmark suite subdivided by logic and mode (sequential or incremental). For the logics, an empty prefix indicates quantified formulas, QF represents quantifier-free, and suffixes indicate the theory (combination). For example:

$$\text{QF\_SLIA} \ = \ \text{quantifier-free strings and linear arithmetic}$$

In sequential mode, formulas have one "`check-sat`" whereas in incremental mode, there are multiple solver queries. There are currently 438,631 non-incremental and 44,333 incremental benchmarks including 84 logics.

2.1.2 *Applications in Academia and Industry*

SMT solvers are important foundations for academic research, including symbolic execution [8, 48], program synthesis [10], solver-aided programming [11], and program verification [12, 13]. Besides these, SMT solvers have also been used for neural network verification [15], type inference [49], and for assisting interactive theorem provers such as Coq and Isabelle [50, 51]. In industry, applications include Microsoft's symbolic execution engine SAGE [52]; SecGuru [18], their tool for verifying network policies; AdaCore Spark [17] for verifying cyber-physical systems; and Zelkova, Amazon Web Services' tool for verifying access policy properties [16, 53].

2.2 AUTOMATED SOFTWARE TESTING

This section first gives basic background on software testing and then specifically on testing SMT solvers.

2.2.1 *Test Generation and Oracle Problem*

The overarching goal of automated software testing is *to increase software reliability and performance*. There are two key problems to achieve this.

Definition 2.2.1 (Test generation & oracle problem)**.** The *test generation problem* is to determine how to fabricate suitable test inputs. The *oracle problem* is to determine how to validate program outputs

Test generation can be mutation-based or generation-based. Mutation-based testing fabricates test inputs by modifying existing inputs, called *seeds*. Generation-based testing fabricates test inputs from scratch, *e.g.*, through grammars or models. Oracles can be differential or metamorphic. Differential oracles compare multiple implementations of the same algorithm. Metamorphic oracles compare a single implementation using *metamorphic relations*, *i.e.*, properties defining how the program output changes when its input is transformed in a specific way. A *tester* is an algorithm that solves the test generation and oracle problem. A *testing campaign* is a continuous effort to identify bugs using a tester. Code coverage is a metric to describe the sufficiency of testing. Line coverage, function coverage, and branch coverage are the most commonly used coverage notions.

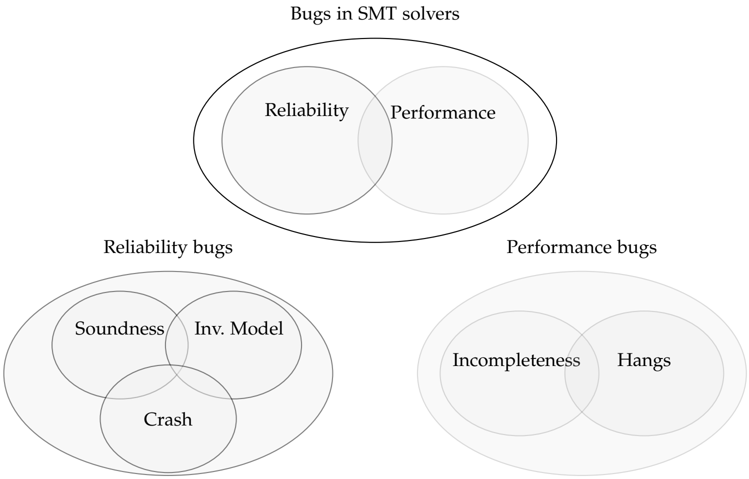


FIGURE 2.1: Taxonomy of bugs in SMT solvers. The bugs in SMT solvers consist of reliability and performance bugs. Reliability break down into soundness, invalid model, and crashes. Performance bugs in turn divide into incompleteness and hangs.

2.2.2 *Bug Types in SMT Solvers*

With a background in software testing, we now address SMT solver testing: (1) bug types, (2) bug reduction, and (3) bug de-duplication.

Definition 2.2.2 (Bug vs bug trigger)**.** A *bug* is a *unique* issue in the program under test. A *bug trigger* is an input that triggers a particular bug. Multiple bug triggers can point to the same bug.

We distinguish between two supertypes of bugs in SMT solvers: Reliability and performance bugs, each with its subtypes (see Figure 2.1).

Definition 2.2.3 (Reliability bugs)**.** Formula φ triggers

- a *soundness bug* if $S(\varphi) \neq \mathcal{O}(\varphi)$.
  - a *refutation soundnesss bug* if $S(\varphi) = \texttt{unsat}$ and $\mathcal{O}(\varphi) = \texttt{sat}$
  - a *solution soundnesss bug* if $S(\varphi) = \texttt{sat}$ and $\mathcal{O}(\varphi) = \texttt{unsat}$
- an *invalid model bug* if the model M returned by the solver does not satisfy φ and does not trigger a soundness bug.
- a *crash bug* if S throws an assertion violation or a segmentation fault.

where S is an SMT solver under test, and $\mathcal{O}$ is a test oracle.

Soundness bugs can have severe consequences on downstream applications. Refutation soundness bugs are considered the most critical, as incorrectly returning `unsat` translates to falsely assuring a property of a software or system.[2] Solution soundness bugs are severe but less severe than refutation soundness bugs. In practice, however, soundness bugs often have both refutation and solution triggers. Invalid model bugs are considered less than soundness and more severe than crash bugs.

Definition 2.2.4 (Performance bugs)**.** Formula φ triggers

- an *incompleteness bug*: if unexpectedly $S(\varphi) =$ `unknown`. We further distinguish two subtypes:
  - *regression incompleteness bug:*
    $S_{old}(\varphi) =$ `sat/unsat` and $S(\varphi) =$ `unknown` where S_{old} is an earlier version of S
  - *implication incompleteness bug:*
    $S(\varphi') =$ `sat` and $S(\varphi) =$ `unknown` if φ' implies φ
    $S(\varphi') =$ `unsat` and $S(\varphi) =$ `unknown` if φ implies φ'
- *hang/performance bug*: if the solver unexpectedly does not solve φ within a given timeout. We call a hang/performance bug a *performance regression* if an earlier version of S was able to solve φ.

The severity of performance bugs depends on the logic and SMT solver developers. Based on our experience the severity is roughly in this order: performance regressions, incompleteness regressions, performance bugs, implication incompleteness bugs.

2.2.3 *Bug detection, Deduplication, and Reduction*

Invalid model and crash bugs are detected by non-zero exit code and matching patterns on standard output and error. The detection of soundness bugs for a metamorphic oracle is direct. For a differential oracle, where two SMT solvers are compared. The solver at fault is detected by validating the model of the solver that returns `sat`. If the model is valid then, the bug is very likely a refutation soundness bug. Otherwise if the model is invalid then it is very likely a solution soundness bug.

[2] Properties are usually formulated as the negations of undesirable states or behaviors.

When testers detect bugs, their triggers are often sizeable, needing further reduction before they can be reported to the developers. We use two program reducers to that end. We use C-Reduce [54], a generic code reduction tool, primarily developed for C but also works for the SMT-LIB language. Besides C-Reduce, we use pydelta [55], a reducer for SMT-LIB.

Multiple bug triggers can point to the same underlying unique bug. Hence, we need de-duplication. To avoid duplicate bug reports on the GitHub issue trackers, we de-duplicate the bugs as follows. Crash bugs are either assertion violations or segmentation faults. We de-duplicate assertion violations via the location information (file name and line number) printed on standard output/error. For soundness and invalid model bugs, we first categorize the bug triggers by theory. We do this because bug triggers in different theories are likely to be unique bugs. Then, we select one bug trigger per theory at a time for reporting. If the bug was fixed, we check the remaining bug-triggering formulas of the same theory. If one of them still triggers a bug in the solver, we repeat until none of them triggers anymore.

3 SEMANTIC FUSION

As of early 2019, there was little reason to doubt the reliability of SMT solvers. Z3 and CVC4, the two most popular and widely used solvers, had been developed for over a decade, and they had few known soundness issues. [1] As a result, Z3 and CVC4 were regarded as mature and were widely trusted. This chapter introduces Semantic Fusion, a novel testing methodology, which challenged the existing trust in the solvers. The results of our testing were surprising and had a significant impact on the field.

3.1 ILLUSTRATIVE EXAMPLES

This section illustrates two instantiations of Semantic Fusion: (1) Sat fusion fuses a pair of satisfiable formulas into a satisfiable formula, and, (2) Unsat fusion fuses a pair of unsatisfiable formulas into an unsatisfiable formula.

SAT FUSION Sat fusion combines two satisfiable formulas into a satisfiable formula. Sat fusion can be described by the following steps: (1) Formula Conjunction, (2) Variable Fusion, and (3) Variable Inversion. Consider the formulas φ_1 and φ_2 in Figure 3.1. The SMT-LIB code represents the following formulas:

$$\begin{aligned}
\varphi_1 &= (x = -1) \wedge (w = (x = -1)) \wedge w \\
\varphi_2 &= (v = (y \neq -1)) \wedge (v \rightarrow \mathit{false}) \wedge (\neg v \rightarrow (y = -1))
\end{aligned}$$

Formula φ_1 is satisfiable since assigning $x = -1$ and $w = \mathit{true}$ satisfies both conjuncts. Formula φ_2 is also satisfiable since we can set y to -1 and v to *false*, which satisfies the formula. In the following, we detail steps 1-3.

STEP 1: FORMULA CONJUNCTION: We conjoin formula φ_1 with formula φ_2 and obtain $\varphi_1 \wedge \varphi_2$ as a result. In the SMT-LIB format, this conjunction is realized by merging variable declaration and assert blocks.

[1] The few existing soundness issues at the time mostly manifested in the theory of Strings which was considered unstable. Besides these, soundness issues in Z3 and CVC4 were rare.

```
; phi1
(declare-fun x () Int)
(declare-fun w () Bool)
(assert (= x  (- 1)))
(assert (= w (= x (- 1))))
(assert w)
(check-sat)
```

```
; phi2
(declare-fun y () Int)
(declare-fun v () Bool)
(assert (= v (not (= y (- 1)))))
(assert (ite v false (= y  (- 1))))

(check-sat)
```

Figure 3.1: Formulas φ_1 and φ_2 in the SMT-LIB format. Shaded: variables to be replaced by inversion terms.

Step 2: variable fusion: We introduce a fresh variable z to fuse the integer variable pairs x in φ_1 and y in φ_2. We define a fusion function: $f(x,y) = x \cdot y$ and construct an equation $z = f(x,y)$. The choice of the fusion function f is determined by the type of the fused variables (c.f. Section 3.2.1). We fuse the occurrences of variables x and y.

Step 3: variable inversion: We dissolve the equation $z = f(x,y)$ to $r_x(y,z) = z \; div \; y$ and $r_y(x,z) = z \; div \; x$, where r_x and r_y are called inversion functions and div denotes the integer division. The purpose of the inversion functions is to recover the original values of x and y. The inversion function $r_x(y,z)$, for example, recovers x by a term that only depends on y and z. We then randomly replace free occurrences of x by $r_x(y,z)$ and free occurrences of y by $r_y(x,z)$. The formula φ_{sat} is by construction satisfiable. The code of φ_{sat} is shown in Figure 3.2.

Why is φ_{sat} satisfiable? Intuitively, because we can construct a model for φ_{sat} from models for φ_1 and φ_2. Let M_1 be a model for φ_1 and M_2 be a model for φ_2. We construct M for φ_{sat} with $M = M_1 \cup M_2 \cup \{z \mapsto M_1(x) \cdot M_2(y)\}$ (see Section 3.2 for details). Formula φ_{sat} in Figure 3.2 is a real case. It triggered a soundness bug in CVC4, which made CVC4 incorrectly report `unsat` on φ_{sat}. We reported this issue to the GitHub CVC4's issue tracker. As per the developers, this was a regression introduced by recent code changes, and they promptly fixed the bug.

Unsat fusion Unsat fusion combines two unsatisfiable formulas into an unsatisfiable formula. We describe the idea behind Unsat fusion in four steps: (1) Formula Disjunction, (2) Variable Fusion, (3) Variable Inversion, and (4) Adding Fusion Constraints. While steps (1), (2), and (3) are similar

```
; CVC4 #3413
(declare-fun v () Bool)
(declare-fun w () Bool)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
; phi1 part
(assert (= (div z y) (- 1)))
(assert (= w (= x (- 1))))
(assert w)
; phi2 part
(assert (= v (not (= y (- 1)))))
(assert (ite v false (= (div z x) (- 1))))
(check-sat)
```

FIGURE 3.2: Fused formula φ_{sat} triggering a soundness bug in CVC4.

to those in Sat fusion, Unsat fusion needs an additional fourth step to ensure the unsatisfiability of the fused formula. Consider the formulas φ_3 and φ_4 in Figure 3.3:

$$\varphi_3 = ((1.0 + x) + 6.0) \neq (7.0 + x)$$
$$\varphi_4 = (0 < y < v \leq w) \land (w/v < 0)$$

Formula φ_3 is trivially unsatisfiable. Formula φ_4 is unsatisfiable, since the left part of the conjunction requires both w and v to be non-negative but the right part requires w and v to be of opposite signs. First, we disjoin the two formulas. We then again choose a pair of free variables in each formula, *e.g.*, variable x in φ_1 and variable y in φ_2, and introduce a fresh variable z and fusion function $f(x,y) = x \cdot y$ with $z = f(x,y)$. We dissolve the equation $z = f(x,y)$ to inversion functions $r_x(y,z) = z/y$ and $r_y(x,z) = z/x$, and randomly substitute the first occurrence of x by $r_x(y,z)$ and both occurrences of y by $r_y(x,z)$.

STEP 4: ADD FUSION CONSTRAINTS: We add $z = f(x,y)$, $x = r_x(y,z)$ and $y = r_y(x,z)$ to the fused formula. We call them *fusion constraints*. Since random substitutions may render the fused formula satisfiable, we need the fusion constraints to ensure that r_x and r_y recover x and y (see Section 3.2 for details). The SMT-LIB code of the resulting fused formula is shown in Figure 3.4.

```
; phi3
(declare-fun x () Real)
(assert (not (= (+ (+ 1.0  x ) 6.0)
                (+ 7.0 x))))
(check-sat)
```

```
; phi4
(declare-fun y () Real)
(declare-fun w () Real)
(declare-fun v () Real)
(assert (and (<  y  v) (>= w v)
        (< (/ w v) 0) (>  y  0)))
(check-sat)
```

Figure 3.3: Formulas φ_3 and φ_4 in the SMT-LIB format. Shaded: variables to be replaced by inversion terms.

```
; Z3 #2391
(declare-fun v () Real)
(declare-fun w () Real)
(declare-fun x () Real)
(declare-fun y () Real)
(declare-fun z () Real)
(assert (or
(not (= (+ (+ 1.0  (/ z y) ) 6.0)   ; phi3 part
        (+ 7.0 x)))
(and (<  (/ z x)  v) (>= w v)       ; phi4 part
(< (/ w v) 0) (>  (/ z x)  0))))
(assert (= z (* x y)))  ; fusion constraint
(assert (= x (/ z y)))  ; fusion constraint
(assert (= y (/ z x)))  ; fusion constraint
(check-sat)
```

Figure 3.4: Fused formula φ_{unsat} of φ_3 and φ_4 triggering a soundness bug in Z3

The fused formula φ_{unsat} in Figure 3.4 has triggered a soundness bug in Z3, *i.e.*, Z3 reports sat on φ_{unsat}, which is incorrect since the formula is unsatisfiable by construction. This bug is only triggered by the fused formula; it cannot be triggered by either of the seed formulas nor by the disjunction/conjunction of the two seed formulas φ_3 and φ_4.

3.2 SEMANTIC FUSION

This section presents Semantic Fusion and how we apply it to stress-testing SMT solvers. We present formal definitions for Sat, Unsat, and Mixed fusion along with proofs, exemplary fusion and inversion functions, and details on YinYang's implementation.

DEFINITIONS We consider first-order logic formulas of the satisfiability modulo theories (SMT). For such a formula φ, we denote the set of free variables by $\mathit{vars}(\varphi)$. A substitution of a variable x in $\mathit{vars}(\varphi)$ by an expression e is denoted by $\varphi[e/x]$. A model M for φ is a function that maps all free variables $x_1, \cdots x_n \in \mathit{vars}(\varphi)$ to values in their respective domains such that $\varphi[M(x_1)/x_1, \cdots, M(x_n)/x_n]$ simplifies to *true*. The model count of a formula φ is $C(\varphi) = |\{M \mid M \models \varphi\}|$. Formula φ is satisfiable if $C(\varphi) \geq 1$ and unsatisfiable otherwise. $\varphi[e/x]_R$ denotes the formula where some of the occurrences of x (possibly none) in φ are replaced by e. It holds $C(\varphi[e/x]) \leq C(\varphi[e/x]_R)$.

The key idea behind Semantic Fusion is to combine two seed tests into a new test that fuses the structures of its ancestors. Applying this to two formulas φ_1 and φ_2 of same satisfiability.

Definition 3.2.1 (Fusion function)**.** Let φ_1 and φ_2 be formulas, x in $\mathit{vars}(\varphi_1)$, and y in $\mathit{vars}(\varphi_2)$. Let further z be a fresh variable $z \notin \mathit{vars}(\varphi_1) \cup \mathit{vars}(\varphi_2)$. We define

$$z := f(x, y)$$

The function f is called a **fusion function** and z is called a fusion variable.

Intuitively, fusion functions act as anchors in the seed formulas φ_1 and φ_2 connecting the variables x and y in interesting ways. To recover the original values of x and y, we introduce inversion functions.

Definition 3.2.2 (Inversion function)**.** Let φ_1 and φ_2 be formulas and f be a fusion function. For $x \in \mathit{vars}(\varphi_1)$, $y \in \mathit{vars}(\varphi_2)$ and $z = f(x, y)$, we define

$$x = r_x(y, z) \quad y = r_y(x, z)$$

functions r_x and r_y are called **inversion functions**.

As an example fusion function, consider $f(x, y) = x + y$. The corresponding inversion functions for x and y are: $r_x(y, z) = z - y$, and $r_y(x, z) = z - x$. We elaborate on choices for fusion and inversion functions in detail later. The following proposition shows that fusing two satisfiable formulas leads to a fused formula of known satisfiability.

3.2.1 *Sat Fusion*

We next present a proposition that shows how we fuse two satisfiable formulas into a satisfiable formula.

Proposition 3.2.1 (Sat fusion). *Let φ_1 and φ_2 be satisfiable formulas with $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. Let further x in $vars(\varphi_1)$ and y in $vars(\varphi_2)$ be variables. Then, $\varphi_{sat} = \varphi_1[r_x(y,z)/x]_R \wedge \varphi_2[r_y(x,z)/y]_R$ is satisfiable.*

Proof. Let M_1 and M_2 be models for φ_1 and φ_2, respectively. We construct a model M for φ_{sat} as follows:

$$\begin{aligned} M(v) &= M_1(v), \quad \text{for } v \in vars(\varphi_1) \\ M(v) &= M_2(v), \quad \text{for } v \in vars(\varphi_2) \\ M(z) &= f(M_1(x), M_2(y)) \end{aligned}$$

Since $x = r_x(y,z)$ by Definition 3.2.2, $M(x) = M(r_x(y,z))$. Thus, via structural induction we have $M(\varphi_1[r_x(y,z)/x]_R) = M(\varphi_1)$. By M's construction, $M(\varphi_1) = M_1(\varphi_1)$, thus $M \models \varphi_1[r_x(y,z)/x]_R$. Similarly, $M \models \varphi_2[r_y(x,z)/y]_R)$, and hence $M \models \varphi_{sat}$. □

3.2.2 *Unsat Fusion*

Proposition 3.2.1 enables us to fuse two satisfiable formulas and obtain a satisfiable formula as a result. We would also like to fuse unsatisfiable formulas into an unsatisfiable formula. However, we cannot simply fuse two unsatisfiable formulas using Proposition 3.2.1 as the following counterexample shows. Consider the two unsatisfiable formulas

$$\varphi_1 = x > 0 \wedge \boxed{x} < 0 \qquad \varphi_2 = \boxed{y} \neq y$$

with the fusion function $z = x + y$. If we replace the shaded occurrence of x by $y - z$ and y by $x - z$, we get the following formula:

$$(x > 0) \wedge (\boxed{z - y} < 0) \wedge (\boxed{z - x} \neq y)$$

This is a satisfiable formula, *e.g.*, any assignments for x, y and z that satisfy $x > 0$ and $y > z$ realize a model. The problem here is that we can freely choose z that does not necessarily preserve $z = f(x,y)$. To prevent this, we add the constraint $z = f(x,y)$ to the formula. For fusing unsatisfiable formulas, we disjoin the formulas, since this is likely to increase the effort of SMT solvers to prove the formula unsatisfiable.

Proposition 3.2.2 (Unsat fusion). *Let φ_1, φ_2 be unsatisfiable formulas with $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. Let further $x \in vars(\varphi_1)$, $y \in vars(\varphi_2)$ be variables. Then, the formula*

$$\varphi_{unsat} = (\varphi_1[r_x(y,z)/x]_R \vee \varphi_2[r_y(x,z)/y]_R) \wedge z = f(x,y)$$

is unsatisfiable.

Proof. Assume the contrary, *i.e.*, φ_{unsat} was satisfiable. Then either (or both) of the following would be satisfiable:

$$\varphi_1[r_x(y,z)/x]_R \wedge z = f(x,y)$$
$$\varphi_2[r_y(x,z)/y]_R \wedge z = f(x,y)$$

Say $\varphi_1[r_x(y,z)/x]_R \wedge z = f(x,y)$ were satisfiable. The formula $\varphi_1[r_x(y,z)/x]_R \wedge z = f(x,y)$ is equivalent to the formula $\varphi_1[r_x(y,z)/x]_R[f(x,y)/z] \wedge z = f(x,y)$, which, by Definition 3.2.2, is equivalent to $\varphi_1 \wedge z = f(x,y)$. This contradicts the assumption, *i.e.*, the unsatisfiability of φ_1. The case for $\varphi_2[r_y(x,z)/y]_R \wedge z = f(x,y)$ is symmetric. □

3.2.3 *Mixed Fusion*

Proposition 3.2.2 enables to fuse two unsatisfiable formulas into an unsatisfiable formula and complements Proposition 3.2.1. The following proposition shows how to fuse a formula pair of mixed satisfiability *i.e.*, when φ_1 is satisfiable and φ_2 is unsatisfiable without loss of generality.

Proposition 3.2.3 (Mixed fusion). *Let φ_1 be a satisfiable and φ_2 be an unsatisfiable formula with $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. Let further $x \in vars(\varphi_1)$, $y \in vars(\varphi_2)$ be variables. Then:*

$\varphi_{mixed\text{-}sat} = \varphi_1[r_x(y,z)/x] \vee \varphi_2[r_y(x,z)/y]$ *is satisfiable; and*

$\varphi_{mixed\text{-}unsat} = \varphi_1[r_x(x,z)/x] \wedge \varphi_2[r_y(x,z)/y] \ \wedge z = g(x,y)$ *is unsatisfiable.*

The proof for Proposition 3.2.3 can be found in Appendix A.1.

3.3 FUSION AND INVERSION FUNCTIONS

This section details the exemplary fusion and inversion functions (see Figure 3.5), and describes the process of designing fusion functions.

Type	Fusion Function	Variable Inversion Functions		ID
		r_x	r_y	
Int	$x+y$	$z-y$	$z-x$	①
	$x+c+y$	$z-c-y$	$z-c-x$	②
	$x*y$	$z \mathit{div}\ y$	$z \mathit{div}\ x$	③
	$c_1*x+c_2*y+c_3$	$(z-c_2*y-c_3)\ \mathit{div}\ c_1$	$(z-c_1*x-c_3)\ \mathit{div}\ c_2$	④
Real	$x+y$	$z-y$	$z-x$	⑤
	$x+c+y$	$z-c-y$	$z-c-x$	⑥
	$x*y$	z/y	z/x	⑦
	$c_1*x+c_2*y+c_3$	$(z-c_2*y-c_3)/c_1$	$(z-c_1*x-c_3)/c_2$	⑧
String	x *str++* y	*str.substr* z 0 (*str.len* x)	*str.substr* z (*str.len* x) (*str.len* y)	⑨
	x *str++* y	*str.substr* z 0 (*str.len* x)	*str.replace* z x ""	⑩
	x *str++* c *str++* y	*str.substr* z 0 (*str.len* x)	*str.replace* (*str.replace* z x "") c ""	⑪

Figure 3.5: Fusion functions with their corresponding variable inversion functions categorized by types. The coefficients $c_1, \cdots, c_3$ are randomly chosen, and *div* denotes integer division.

exemplary fusion functions Let us consider the Int and Real categories. The first two fusion and inversion functions in these categories are based on addition/subtraction and multiplication/division. When division and multiplication of variables are used as function and inversion functions, a formula in linear logic might become non-linear. This is because we replace free variable occurrences with inversion functions that include the division operator. Another inversion function for real and integer arithmetic is $c_1 * x + c_2 * y + c_3$. The intuition behind $c_1 * x + c_2 * y + c_3$ is to synthesize arbitrary polynomial combinations of the variables x and y with $c_1, \cdots, c_3$ being random coefficients. Let us consider Strings next. In the first row of the String category, we define z as the concatenation of the two strings x and y. Say $x =$ "foo" and $y =$ "bar", then $z = x$ *str++* $y =$ "foobar". We retrieve x by the substring of z from 0 to $|x|$ and for y the substring from $|y|$ to the end of z. Another way to retrieve y is to use the replace function instead of substring. The expression *str.replace* z x "" denotes the replacement of the first occurrence of x in z by the empty string "", which results in "bar". In addition, we can insert a random string c into x *str++* y by x *str++* c *str++* y to make the fusion function more complex, and then retrieve y by replacing

x and c with "" sequentially. Semantic Fusion is not restricted to these fusion and inversion functions of Figure 3.5. A richer set of fusion and inversion functions can be designed based on Definitions 3.2.1 and 3.2.2.

DESIGNING FUSION FUNCTIONS The key idea behind all fusion functions is to construct an equation based on a pair of free variables of two seed formulas and then use the inversion functions to retrieve the original values of the fused variables. This way, we link the structures of seed formulas in a nontrivial and random manner. To design such functions, we began with arithmetic following the natural mathematical intuitions. We then translated such intuitions to other theories such as strings and booleans. A necessary condition is that f is invertible and the inverses with respect to x and y can be represented by SMT-LIB code for the inversion functions. We have generally favored simple, short fusion functions over complicated ones to guarantee correctness. However, more complicated fusion functions may be designed and verified with an SMT solver.

3.4 THE YINYANG TOOL

Based on Semantic Fusion, we designed and engineered the bug detection tool YinYang to stress-test SMT solvers.

ALGORITHM Algorithm 1 presents a parameterized algorithm of YinYang. The main procedure takes the oracle of the seed formulas $\mathcal{O} \in \{\texttt{sat}, \texttt{unsat}\}$, SMT solver under test S, and a set of seed formulas $seeds_{\mathcal{O}}$ as its inputs. Each seed in $seeds_{\mathcal{O}}$ has the same satisfiability as the oracle $\mathcal{O}$. The sets of *incorrects* and *crashes* are used for collecting soundness and crash bugs, respectively, and are both initialized to the empty set. The while loop body is executed until a termination criterion is met, *e.g.*, a timeout or an interrupt by the user (Line 3). We first randomly choose two formulas φ_1, φ_2 from $seeds_{\mathcal{O}}$ and pass them to the `fuse` function along with the oracle $\mathcal{O}$. The `fuse` function returns the fused formula φ_{fused} (Line 6). Then, we check whether the SMT solver S has crashed on solving φ_{fused}. If so, we have found a crash bug and will add φ_{fused} to *crashes*. Otherwise, if S does not crash, we check whether $S(\varphi_{fused})$ is inconsistent with the oracle $\mathcal{O}$ (Line 9). If so, there is a soundness issue and add φ_{fused} to the set *incorrects*.

Algorithm 2 presents the implementation of the `fuse, variable_fusion` function. It takes two seed formulas φ_1 and φ_2 as its inputs and retrieves the sets of their free variables $vars(\varphi_1)$ and $vars(\varphi_2)$, respectively. Then,

Algorithm 1 YinYang's main process

```
procedure YinYang($\mathcal{O}$, $S$, $seeds_{\mathcal{O}}$)
    $incorrects \leftarrow \emptyset,\ crashes \leftarrow \emptyset$
    while no termination criterion met do
        $\varphi_1 \leftarrow random.choice(seeds_{\mathcal{O}})$
        $\varphi_2 \leftarrow random.choice(seeds_{\mathcal{O}})$
        $\varphi_{fused} \leftarrow$ FUSE$(\mathcal{O}, \varphi_1, \varphi_2)$
        if $S(\varphi_{fused}) =$ crash then
            $crashes \leftarrow crashes \cup \{\varphi_{fused}\}$
        else if $S(\varphi_{fused}) \neq \mathcal{O}$ then
            $incorrects \leftarrow incorrects \cup \{\varphi_{fused}\}$
```

Algorithm 2 Semantic Fusion on two SMT formulas

```
function FUSE($\mathcal{O}$, $\varphi_1$, $\varphi_2$)
    $vars(\varphi_1) \leftarrow get\_free\_variables(\varphi_1)$
    $vars(\varphi_2) \leftarrow get\_free\_variables(\varphi_2)$
    $T \leftarrow random\_map(vars(\varphi_1), vars(\varphi_2))$
    $\varphi'_1, \varphi'_2 \leftarrow$ VARIABLEFUSION$(T, \varphi_1, \varphi_2)$
    if $\mathcal{O} =$ sat then
        return $\varphi'_1 \wedge \varphi'_2$
    else
        $\varphi' \leftarrow \varphi'_1 \vee \varphi'_2$
        for $(z, x, y) \in T$ do
            $\varphi' \leftarrow \varphi' \wedge (z = f(x, y))$
        return $\varphi'$

function VARIABLEFUSION($T$, $\varphi_1$, $\varphi_2$)
    $\varphi'_1 \leftarrow \varphi_1\ \varphi'_2 \leftarrow \varphi_2$
    for $(z, x, y) \in T$ do
        $\varphi'_1 \leftarrow \varphi'_1[r_x(y, z)/x]_R$
        $\varphi'_2 \leftarrow \varphi'_2[r_y(x, z)/y]_R$
    return $\varphi'_1, \varphi'_2$
```

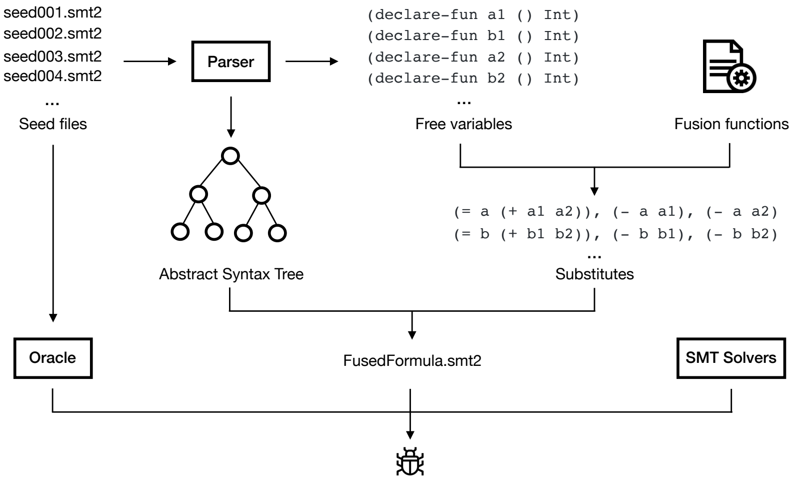


FIGURE 3.6: The YinYang framework schematically: the seed formulas of known oracle get parsed to an AST representation. YinYang reads the fusion functions and inversion functions from a configuration file. Out of these, the substitute triplets are generated which are used for fusion.

we create random triplets T, for $(z, x, y) \in T$ where $x \in vars(\varphi_1)$ and $y \in vars(\varphi_2)$, and z is the fresh variable. In the `variable_fusion` function, we substitute randomly chosen occurrences of x in φ_1 and y in φ_2 by the inversion function terms $r_x(y, z)$ and $r_y(x, z)$ from Figure 3.5. If oracle $\mathcal{O}$ is *sat*, we perform Sat fusion (Proposition 3.2.1) and return the conjunction of φ_1' and φ_2' directly (Line 12). If oracle $\mathcal{O}$ is *unsat*, we perform Unsat fusion (Proposition 3.2.2), *i.e.*, we disjoin φ_1' and φ_2', and add a fusion constraint for each triplet $(x, y, z) \in T$ (Lines 15-17) and return the result. In principle, YinYang guarantees the absence of false positives, given that the seed formulas $seeds_{\mathcal{O}}$ are correctly labeled. In practice, however, the solvers may report `unknown`, which could be either seen as a crash or ignored.

IMPLEMENTATION OF YINYANG After the first publication of Semantic Fusion at PLDI 2020, we re-built YinYang in a total of 12,117 lines of Python 3.7 code. We reconstructed the architecture of YinYang and added several features. The framework is open-sourced on GitHub. [2] Figure 3.6 shows the framework of YinYang. As inputs, the framework takes seed files of the

[2] https://github.com/testsmt/yinyang

same satisfiability and one or more SMT solvers under test. The seed files are parsed to an abstract syntax tree and free variables are pre-computed. The framework reads a configuration file of which it generates the 3-tuple to perform the substitutions. The 3-tuples consist of the fusion function and the two inversion functions, each formulated as an equation with hard-coded variables x, y, z. By randomly substituting free variables in the abstract syntax tree and concatenating two formulas, we finally get the fused formula. If an SMT solver reports a different satisfiability with the seeds, a bug will be reported. Users can customize the command-line interface of YinYang and also specify custom fusion and inversion functions. YinYang accepts SMT solver binaries as test targets and obtains the solving results from the stdout stream, which makes YinYang compatible with most SMT solvers. When there are multiple fusion/inversion function choices for f, r_x and r_y, YinYang makes a random choice. We designed a DSL similar to the SMT-LIB language for configuring fusion functions. The following code shows the syntax for realizing fusion function ②:

```
#begin
(declare-const x Int)(declare-const y Int)
(declare-const z Int)(declare-const c Int)
(assert (= z (+ (+ x y) c)))
(assert (= x (- (- z y) c)))
(assert (= y (- (- z x) c)))
#end
```

In our DSL, each triplet of a fusion and two inversion functions is surrounded by a "#begin-#end" block. Variables x,y, and z are hard-coded to represent free variables of the first and second seed and the new variable in the fusion function respectively. They are all declared as three constants. An additional constant "c" represents a randomly chosen constant. The first assert represents the equation $z := f(x, y)$, the second and third represent the equations $x = r_x(y, z)$ and $y = r_y(x, z)$ respectively. YinYang will read the fusion function from the template while fusing the formulas.

3.5 EMPIRICAL EVALUATION

This section presents the details of our extensive evaluation of YinYang, demonstrating the practical effectiveness of the Semantic Fusion methodology. Between June and October 2019, we ran YinYang to test the default arithmetic and string solvers of Z3 and CVC4. We chose Z3 and CVC4 since they (1) are popular and widely used in academia and industry, (2) support a rich set of logics, and (3) adopt an open-source development model. During our testing period, we filed numerous bugs on their GitHub issue trackers. This section describes the outcome of our testing effort.

RESULT SUMMARY AND HIGHLIGHTS

- *Many confirmed bugs:* In four months, YinYang found 62unique bugs in Z3 and CVC4. Out of these, 57 were already fixed by the developers.
- *Many soundness bugs:* YinYang found 34 soundness bugs in Z3 and 5 in CVC4. These represent 16% of the reported Z3 soundness bugs of the 2014 - 2019 and 11% of the reported CVC4 soundness bugs 2011 - 2019. Some of the bugs affect multiple historical release versions.
- *Bugs in various logics:* YinYang found bugs in various logics, *e.g.*, QF_NRA, QF_NIA, NRA, NIA, QF_S, and QF_SLIA. Most of the bugs in Z3 were found QF_SLIA (20) and NRA (16) and while most of the bugs in CVC4 were found in QF_S (5).

3.5.1 *Evaluation Setup*

HARDWARE SETUP Since July 2019, YinYang tested Z3 and CVC4 on three machines. The first machine is equipped with an Intel Xeon CPU E5-2680 28-core processor and 256GB RAM. The second machine is equipped with an Intel Core i7-8700 6-core processor and 16GB RAM. The third machine has an AMD Ryzen Threadripper 2990WX processor with 32 cores and 32GB RAM. The OS on all three machines is Ubuntu 18.04 64-bit.

TEST SEED FORMULAS Figure 3.7 shows the formula counts of the respective benchmarks that we used. The majority of the seed formulas come from the SMT-LIB benchmark suite maintained by the SMT-LIB Initiative [47]. We chose the SMT-LIB benchmarks as our test seeds since they make the largest collection of SMT formulas in the SMT-LIB 2.6 language. The SMT-LIB benchmarks are also used in the SMT Competition. Therefore, these formulas are unlikely to trigger bugs in Z3 and CVC4 since they have already been run on them. This helps us isolate the effects of Semantic Fusion. We choose the following logics: LIA, LRA, NRA, QF_LIA, QF_LRA, QF_NRA, QF_SLIA, and QF_S. L represents linear, N represents non-linear, IA represents integer arithmetic, RA represents real arithmetic, QF represents quantifier-free, and S represents string logic. Besides the SMT-LIB benchmarks, we also used the benchmarks from StringFuzz [56].

Benchmark	#UNSAT	#SAT	Total
LIA	203	139	342
LRA	1,316	714	2,030
NRA	3,798	-	3,798
QF_LIA	1,191	1,318	2,509
QF_LRA	384	522	906
QF_NRA	4,660	4,751	9,411
QF_SLIA	5,492	22,657	28,149
QF_S	6,390	12,561	18,951
StringFuzz	4,903	4,098	9,001

FIGURE 3.7: The formula counts of the respective benchmarks.

The StringFuzz benchmarks only includes formulas of QF_S logic. They do not trigger any bugs in the latest versions of Z3 and CVC4. We preprocessed all formulas (from the SMT-LIB benchmarks and StringFuzz) with Z3 to subdivide them into a satisfiable and an unsatisfiable set. We cross-checked with CVC4 to ensure the correctness of these ground truths. In total, we obtained 75,097 seed formulas, 46,760 of which are satisfiable and 28,337 are unsatisfiable. We give detailed statistics on the test seed formulas in Figure 3.7. There is only one available satisfiable formula for NRA logic in the SMT-LIB benchmark, hence we omit this logic.

SMT SOLVERS We selected the SMT solvers Z3 and CVC4 for the evaluation of YinYang. We chose them because:

- Z3 and CVC4 are the two most popular SMT solvers. They are mature and widely used in academia and industry.
- Z3 and CVC4 have state-of-the-art performance. Both regularly rank high in the annual SMT competition.
- Z3 and CVC4 support most of the features and logics in the SMT-LIB standard, while the other SMT solvers only partially support the SMT-LIB standard.
- Z3 and CVC4 have open-source issue trackers on GitHub, and their developers are active and responsive. This helps our testing effort as we can quickly get feedback, and filed bugs are fixed promptly.

For CVC4, we use its `--strings-exp` option to enable support for the string logic and default configuration for the other logics. For Z3, we use both the default string solver z3str3.[3] We compiled both solvers with assertions enabled. Both solvers are compiled with assertion enabled.

BUG REDUCTION When a bug is found, we reduce the fused formula to a small enough size for reporting. We use C-Reduce [54], a C code reduction tool, which also works for the SMT-LIB language. We implemented a pretty printer to help with the bug reduction process, *i.e.*, when C-Reduce has converged to a still very large formula or hanged. The pretty-printer makes modifications to the AST of a formula, *i.e.*, flattens nestings of the same operator, removes additions and multiplications with neutral elements, and returns the modified formula in a human-readable format.

3.5.2 *Quantitative Evaluation*

We guide our quantitative evaluation by the following five consecutive research questions:

RQ1: How many bugs can YinYang find?

RQ2: How significant are the bug-finding results?

RQ3: Can YinYang improve code coverage?

RQ4: Is Semantic Fusion necessary for finding bugs?

RQ5: Which fusion functions caused YinYang's soundness bug findings?

RQ1: How many bugs can YinYang find?

From July 2019 to October 2019, we extensively tested Z3 and CVC4 with YinYang. YinYang usually reports many bug-triggering test cases in one testing round. To avoid duplicate bug reports, we always use the trunk versions of the solvers for testing. Once the developers have fixed a bug, we validate the fixed version on the rest of the formulas that triggered bugs in the previous testing round. If the solvers passed all formulas and no bug was triggered, we started a new testing round. During our four months of testing, YinYang generated around 800 million test formulas. On average, YinYang generates 41.5 test formulas per second when run in the single-threaded mode. Figure 3.8a shows the bug counts categorized by reported,

[3] `smt.string_solver=z3str3`

Status	Z3	CVC4	Total
Reported	62	14	76
Confirmed	51	11	62
Fixed	48	9	57
Duplicate	4	1	5
Won't fix	1	1	2

(a)

Type	Z3	CVC4	Total
Soundness	34	5	39
Crash	12	5	17
Invalid model	3	0	3
Other	2	1	3

(b)

Logic	Z3	CVC4	Total
QF_SLIA	20	5	25
NRA	16	0	16
QF_S	8	3	11
QF_NIA	2	3	5
QF_NRA	4	0	4
NIA	1	0	1

(c)

FIGURE 3.8: (a) Status of the reported bugs in Z3 and CVC4, (b) types of the confirmed bugs in Z3 and CVC4, and (c) affected SMT logics of the confirmed bugs in Z3 and CVC4.

confirmed, fixed, duplicate, and won't fix. From the 76 reported bugs, 62 bugs were confirmed by the developers as real bugs and 57 bugs were fixed. Although we devoted equal testing effort to both solvers, YinYang found more bugs in Z3 (51 confirmed bugs) and clearly fewer bugs in CVC4 (11 confirmed bugs). Having observed that YinYang can find a significant number of bugs in Z3 and CVC4, Figure 3.8b shows the bug type overview of YinYang's findings. We distinguish the following three types of bugs:

- *Soundness bugs*: A formula triggers a soundness bug if the solver reports an incorrect solving result.
- *Crash bugs*: A formula triggers a crash bug if the solver terminates abnormally or throws internal errors while processing the formula.
- *Performance and unknown bugs*: A formula triggers a performance bug if the solver reports unknown or cannot terminate on a simple formula and the developers confirm implementation issues.

Overall, the most common bug category is for soundness bugs (34 out of the 62 confirmed bugs) followed by crash bugs (12 out of the 62 confirmed bugs). This is consistent for both solvers, which shows the strength of YinYang in finding soundness bugs. Although we designed YinYang to target soundness and crash bugs, we also considered performance bugs. We have found these bugs during the reduction process of C-Reduce. As performance bugs are less interesting than soundness and crash bugs, we stopped reporting performance bugs after several bug reports and solely focused on soundness and crash bugs subsequently. Figure 3.8c shows the logic distribution among the confirmed bugs. In Z3, we found most of the

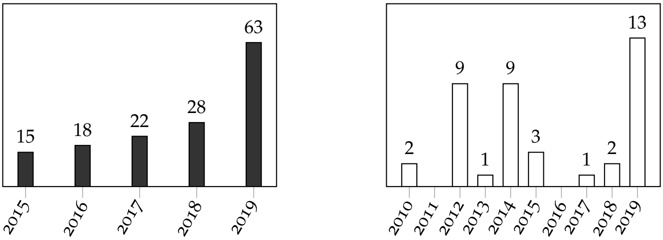


Figure 3.9: Soundness bugs in Z3 (left) and CVC4 (right) per year.

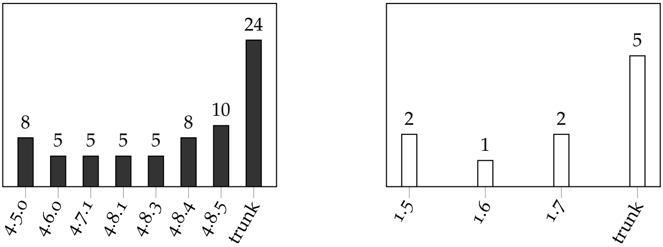


Figure 3.10: Soundness bug finding affecting Z3 (left) and CVC4 (right).

bugs in QF_SLIA (20) followed by NRA (16) and QF_S (8). In CVC4, we found most of the bugs in QF_SLIA (5) followed by QF_NIA (3).

RQ2: How significant are the bug-finding results?

To approach this question, we consider the most critical bugs in SMT solvers, *i.e.*, the soundness bugs. Soundness bugs in SMT solvers are rare and heavily penalized when detected in the SMTComp competitions. We have conducted a study on soundness bugs based on the GitHub issue trackers of Z3 and CVC4. The results are shown in Figure 3.9. For Z3, we considered April 2015 as the start date, right after Z3 was released on GitHub. For CVC4, we have data since July 2010 as CVC4's previous Bugzilla issue tracker was migrated to GitHub. Z3 supports many logics and has become popular on GitHub. As the end date, we set July 2019.

However, there were only 146 soundness bugs reported on the Z3 issue tracker from April 2015 to October 2019. For CVC4 this number is even lower. Since July 2010, there were only 42 soundness bugs. As an intermediate conclusion to RQ2, we state that YinYang has found a significant number of the soundness bugs in both Z3 and CVC4. To more deeply understand the significance of our soundness bug-findings, we studied the influence of soundness bugs in different releases of Z3 and CVC4. Figure 3.10 shows the results. We selected all released versions of Z3 and CVC4 that support the formulas triggering soundness bugs. Z3 4.5.0 was released on November 8, 2016, and CVC4 1.5 was released on July 10, 2017, which means that YinYang found 8 soundness bugs in Z3 that were latent for 3 years, and 2 soundness bugs in CVC4 that were latent for 2 years. YinYang has found long-latent bugs missed by solver developers, users, regression testing, and prior automated testing. This confirms the significance of our bug findings.

RQ3: Can YinYang improve code coverage?

In RQ3, we have observed that the reason for our bug-finding ability is likely to be variable fusion. Therefore we may ask: *What are the unique features of variable fusion? Do we cover different code inside the SMT solvers?* In this research question, we use code coverage, a standard evaluation metric for software testing, to understand whether YinYang can cover additional code inside the SMT solvers. To investigate the coverage improvement of YinYang, we consider the following steps:

1. Run Z3 and CVC4 on all formulas on each benchmark.
2. Measure the line, function, and branch coverage of the solvers. The results are labeled as *Benchmark*.
3. After running Z3 and CVC4 on each benchmark, run YinYang for one hour in single-threaded mode on each benchmark.
4. Measure the line, function, and branch coverage of the solvers. The results are labeled as *YinYang*.

The timeout for the solvers is set to two seconds. For measuring coverage, Z3 and CVC4 are compiled in debug mode and without optimizations. The coverage measurement tool is Gcov [57]. Figure 3.11 shows the results. The numbers represent the percentages (%) of lines (l), functions (f) and branch (b) covered respectively. The highest coverages are shaded. Since SMT solvers have large code bases (Z3 has over 436K LOC, CVC4 has over 238K LOC), 1% line coverage improvement translates to thousands of additionally covered lines. First, we observe that both Z3 and CVC4 mostly achieve less than 30% line, function and branch coverage. This may

		LIA						LRA					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	10.6	14.6	3.4	9.4	13.2	2.8	10.4	14.5	3.3	9.8	13.8	3.1
	YinYang	12.0	16.7	3.8	14.5	18.6	4.9	13.4	17.3	4.3	13.3	16.8	4.3
CVC4	*Benchmark*	14.7	29.1	5.4	15.3	28.9	6.0	14.2	27.3	5.3	12.8	24.9	4.6
	YinYang	16.4	31.4	6.2	17.5	31.9	7.2	16.1	30.6	6.4	15.7	29.4	6.2

		NRA						QF_NRA					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	-	-	-	10.6	13.1	3.5	12.0	13.5	4.6	11.3	13.0	4.1
	YinYang	-	-	-	11.9	14.8	3.9	13.4	15.3	5.1	12.5	14.5	4.6
CVC4	*Benchmark*	-	-	-	16.0	30.9	6.2	12.8	22.6	4.7	13.4	22.6	5.0
	YinYang	-	-	-	17.9	33.9	7.2	15.2	28.7	5.8	15.7	27.8	6.2

		QF_LIA						QF_LRA					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	11.5	16.3	3.7	13.3	17.6	4.3	7.8	12.6	2.6	7.3	11.1	2.4
	YinYang	14.8	19.8	5.1	16.1	20.6	5.5	14.7	18.7	5.2	14.3	17.8	5.2
CVC4	*Benchmark*	11.0	20.2	3.5	11.6	20.2	3.8	10.7	19.9	3.3	10.6	19.6	3.4
	YinYang	12.7	23.8	4.4	12.3	20.9	4.1	13.7	27.2	5.1	12.9	25.6	4.6

		QF_SLIA						QF_S					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	10.2	15.2	3.4	11.5	15.4	4.0	12.5	17.7	4.3	11.8	16.2	4.0
	YinYang	10.4	15.5	3.5	13.1	16.9	4.7	12.6	17.8	4.3	12.8	17.0	4.4
CVC4	*Benchmark*	15.2	25.3	6.8	18.4	32.6	7.4	16.2	28.6	6.5	14.5	26.2	5.6
	YinYang	16.2	26.5	7.3	19.6	34.0	8.0	16.6	29.9	6.7	14.8	26.3	5.9

		StringFuzz					
		SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	13.4	18.3	4.6	12.7	17.5	4.2
	YinYang	13.7	18.4	4.8	13.6	18.3	4.7
CVC4	*Benchmark*	19.6	36.3	8.2	20.3	35.9	9.1
	YinYang	20.0	36.6	8.5	20.8	36.2	9.5

FIGURE 3.11: Coverage evaluations. The numbers represent the percentage (%) coverage for the corresponding coverage metric. Column *l*, *f*, *b* represent line coverage, function coverage, and branch coverage respectively. Higher coverage between *Benchmark* and *YinYang* is shaded.

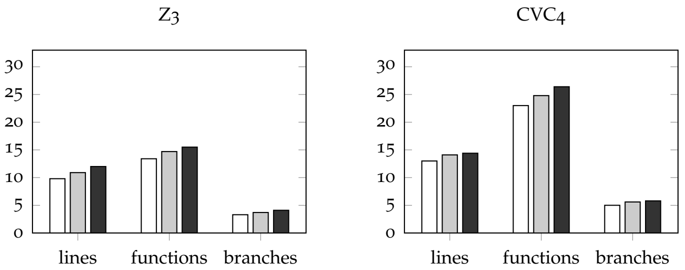


FIGURE 3.12: Coverage improvement (%) of ConcatFuzz (in gray) and YinYang (in black) over Benchmark (in white) averaged over all logics.

be explained by the many mutually exclusive features that CVC4 and Z3 support. The main observation is that YinYang can consistently increase the coverage achieved by the *Benchmark*. This indicates that YinYang can enhance benchmark formulas and exercise previously uncovered code. Furthermore, YinYang can achieve this noticeable coverage improvement in only one hour.

RQ4: Is Semantic Fusion necessary for finding bugs?

This research question investigates whether we can obtain our bug findings with a simpler approach. As mentioned earlier, Semantic Fusion consists of two main steps: (1) formula concatenation and (2) variable fusion and inversion. Step (2) is the core technique of Semantic Fusion. Let ConcatFuzz be the simple concatenation tool where we solely perform step (1) and disable step (2), *i.e.*, ConcatFuzz only combines formulas by conjunction (for satisfiable formulas) and disjunction (for unsatisfiable formulas) without variable fusion and inversion. To see whether Semantic Fusion is necessary for triggering bugs, we ran ConcatFuzz on the ancestor seeds of 50 reported bugs that YinYang found. In only 5 out of 50 cases, ConcatFuzz was able to retrigger the bug. This indicates that simple formula concatenation is unable to trigger most of the bugs found by YinYang, which shows the necessity of the core technique of Semantic Fusion. In addition, we also repeated the code coverage evaluation of RQ3 to understand the code coverage difference between ConcatFuzz and YinYang. Figure 3.12 shows the code coverage of ConcatFuzz and YinYang averaged over all logics. The results show that both YinYang and ConcatFuzz consistently achieve higher line, function, and branch coverage than the Benchmark. The coverage improvement of ConcatFuzz over the benchmark, partially explains why ConcatFuzz can retrigger some of the bugs. However, the results also reflect that the average code coverage of YinYang dominates ConcatFuzz. YinYang achieves on average 1.1% more lines (approx. 2,800 lines) in Z3 and 0.3% (approx. 480 lines) in CVC4, which can partially explain why YinYang can trigger more bugs than ConcatFuzz. In summary, our results show that semantic fusion is necessary for YinYang's effectiveness, and code coverage and bug count correlate.

RQ5: Which fusion functions caused YinYang's soundness bug findings?

Having observed that Semantic Fusion is necessary for bug finding, we next investigate the individual contribution of each fusion function. To that end, we again used the soundness bugs from RQ1 for which we saved the seeds and tried to retrigger them. Due to the randomness of Semantic Fusion, not all soundness bugs can be retriggered in a short time frame. After filtering out the bugs that cannot stably reproduce within 30 minutes, 20 soundness bugs remained. For each bug, we then gradually shrank the fusion functions used in YinYang to a minimal set. The resulting fusion functions in this minimal set all contribute to finding the corresponding bug. To reduce fusion functions, we used the following procedure. First, we delete all fusion functions that are not applied to the seed formulas. These fusion functions certainly do not contribute to the bug finding. Second, we shrink the fusion functions with non-linear operators if possible. Non-linear fusion functions might turn linear formulas into a non-linear formula the logic of the original seeds. We hence attempted to remove all the unnecessary non-linear fusion functions to isolate the effects of linear fusion functions. Third, we shrink the fusion functions with constants. Functions without constants are special cases of the fusion function with constants. The rationale behind this is: if the more specific fusion functions still retrigger the bug, then the corresponding general fusion functions are unnecessary. Finally, we delete the remaining fusion functions one by one. The resulting set of fusion functions is locally minimal, *i.e.*, none of the fusion functions can be removed. Figure 3.13 shows the results of the 20 soundness bugs after shrinking. We first observe that 6 out of 11 fusion functions contribute to soundness bug findings (① ③ ⑦ ⑨ ⑩ ⑪). Fusion functions that do not contribute to any of the bugs are the general cases for integers and reals (② ④ ⑥) and the real addition (⑤). Fusion functions ① ③ ⑦ ⑨ ⑩ ⑪ contribute individually, while bugs #2450, #2514 and #2516 have to be

Bug ID	Solver	Satisfiability	Logic of seeds	Logic	Fusion functions
#2372	Z3	`unsat`	NRA	NRA	①
#2391	Z3	`unsat`	QF_LRA	QF_NRA	⑦
#2422	Z3	`sat`	QF_S	QF_S	⑨
#2450	Z3	`unsat`	LRA	NRA	① ⑦
#2483	Z3	`unsat`	NRA	NRA	①
#3203	CVC4	`unsat`	QF_SLIA	QF_SLIA	①
#2513	Z3	`unsat`	QF_SLIA	QF_S	①
#2514	Z3	`unsat`	QF_SLIA	QF_SNIA	③ ⑩
#2516	Z3	`unsat`	QF_SLIA	QF_SLIA	③ ⑩
#3217	CVC4	`unsat`	QF_SLIA	QF_SLIA	-
#2530	Z3	`unsat`	NRA	NRA	①
#2531	Z3	`unsat`	QF_SLIA	QF_S	⑩
#2533	Z3	`unsat`	QF_SLIA	QF_SLIA	-
#3272	CVC4	`sat`	QF_SLIA	QF_SLIA	⑩
#2573	Z3	`unsat`	LRA	NRA	⑦
#2613	Z3	`unsat`	QF_SLIA	QF_SLIA	⑪
#3357	CVC4	`unsat`	QF_S	QF_S	-
#2618	Z3	`unsat`	QF_S	QF_S	⑪
#2632	Z3	`sat`	QF_S	QF_S	⑨
#3412	CVC4	`sat`	QF_SLIA	QF_NIA	③

FIGURE 3.13: Soundness bugs that can be re-triggered from the seeds within 30 minutes. The function IDs in the last column refer to a minimal set of fusion functions for re-triggering each bug. '-' refers to bugs that could be found without fusion functions.

triggered by a combination of two fusion functions. In 8 out of 20 cases, string fusion functions were necessary. In 5 out of 20 cases, non-linear fusion functions were necessary and in 5 out of 20 the linear fusion function ① was exclusively necessary. As a conclusion, this experiment showed that the majority of fusion functions are necessary for bug finding, especially the simple ones for reals and ints and all the string fusion functions.

3.6 SELECTED BUGS

This section presents a selection of bugs that we found in CVC4 and Z3. We have found soundness bugs, segmentation faults, assertion violations, and performance issues in multiple logics. The original seed formulas φ_{fused} that trigger the bugs are too large to be presented. We therefore present the reduced formulas which we have obtained from bug reduction on the original bug-triggering formulas.

figure 3.14a shows an unsatisfiable formula in the string logic QF_S. The formula has two asserts. The second assertion demands variable a to be the concatenation of b and c. The first assertion includes a query on whether c is matched by regex "aa"*. This is conjoined with a check on whether c equals "0" (see lines 8 to 11). The formula is unsatisfiable since the two conditions contradict each other. However, Z3 reports sat on Formula 3.14a, which is incorrect.

figure 3.14b shows an unsatisfiable formula in the non-linear real arithmetic logic QF_NRA. Z3 reported sat on this formula and gave the following incorrect model:

```
(define-fun e () Real 1.0)
(define-fun f () Real 2.0)
(define-fun a () Real 1.0)
(define-fun b () Real (- 1.0))
(define-fun c () Real 0.0)
(define-fun d () Real 1.0)
```

This model does not satisfy the formula. It causes conflicts between two constraints — the first is the constraint on line 10, while the second is on line 11. According to the SMT-LIB standard, an arbitrary but consistent value v may be chosen for such division-by-zero predicates. Thus, the formula is unsatisfiable. However, Z3 reported sat on the formula. Z3 chose a positive f, and therefore v has to be positive contradicting line 11.

figure 3.14c shows an unsatisfiable formula in the string logic QF_S. Z3 incorrectly reported sat on this formula and gave an incorrect model. The developers made some major changes to fix this bug — 28 files with 486 additions and 144 deletions were necessary to fix it. The bug is triggered by incorrect implementations suffixof and prefixof.

figure 3.14d also shows a formula in the string logic QF_S. The formula is unsatisfiable but CVC4 incorrectly reported sat on it. The formula has been reduced from the same original test case as Formula 3.14a which also triggered a Z3 soundness bug. CVC4 and Z3 both report sat on the original formula. These two bugs show the benefits of our approach over differential testing. In this case, differential testing would not be able to capture these bugs as the results from both

```
(declare-fun a () String)
(declare-fun b () String)
(declare-fun c () String)
(assert (and (str.in.re c
(re.* (str.to.re "aa")))
(= 0  (str.to.int
(str.replace a b
(str.at a (str.len a)))))))
(assert (= a (str.++ b c)))
(check-sat)
```

(a) Soundness bug in Z3: Z3 returns `sat` on this unsatisfiable QF_S formula.

https://github.com/Z3Prover/z3/issues/2618

```
(declare-fun a () Real)
(declare-fun b () Real)
(declare-fun c () Real)
(declare-fun d () Real)
(declare-fun e () Real)
(declare-fun f () Real)
(assert (and (> 0 (- d f))(= d (ite
(>= (/ a c) f) (+ b f) f))(> 0 (/ a
(/ c e)))(or (= e 1.0) (= e 2.0))
(> d 0) (= c 0)))
(check-sat)
```

(b) Soundness Bug in Z3: Z3 reported `sat` on this unsatisfiable QF_NRA formula.

https://github.com/Z3Prover/z3/issues/2391

```
(declare-fun a () String)
(declare-fun b () String)
(declare-fun c () String)
(declare-fun d () String)
(assert (= a (str.++ b d)))
(assert (or (and
(= (str.indexof
(str.substr a 0
(str.len b)) "=" 0) 0)
(= (str.indexof b "=" 0) 1))
(not (= (str.suffixof "A" d)
(str.suffixof "A"
(str.replace c c d))))))
(check-sat)
```

(c) Soundness bug in Z3: Z3 reports `sat` on this unsatisfiable QF_S formula.

https://github.com/Z3Prover/z3/issues/2513

```
(declare-const a String)
(declare-const b String)
(declare-const c String)
(declare-const d String)
(declare-const e String)
(declare-const f String)
(assert (or (and (= c (str.++ e d))
(str.in.re e (re.* (str.to.re "aaa")))
(> 0 (str.to.int d)) (= 1 (str.len e))
(= 2 (str.len c))) (and (str.in.re f
(re.* (str.to.re "aa"))) (= 0
(str.to.int (str.replace (str.replace
a b "") "a" ""))))))
(assert
    (= a (str.++ (str.++ b "a") f)))
(check-sat)
```

(d) Soundness bug in CVC4 reporting `sat` on this unsatisfiable QF_S formula.

https://github.com/CVC4/CVC4/issues/3357

```
(declare-fun a () String)
(declare-fun b () String)
(assert (= (str.++ (str.substr "1" 0
(str.len a)) "0") b))
(assert (< (str.to.int b) 0))
(check-sat)
```

(e) Soundness Bug in Z3: This bug has inspired Z3 developers to implement new rewrites in Z3.

https://github.com/Z3Prover/z3/issues/4138

```
(declare-fun a () String)
(assert (= (str.at (str.substr
(str.substr a 1 (- (str.len a) 1)) 1
(- (str.len (str.substr a 1 (-
(str.len a) ))) 1)) 0) "\f"))
(check-sat)
```

(f) Invalid Model Bug in Z3: This bug has led to a discussion on finding rewrites automatically via fuzzers.

https://github.com/Z3Prover/z3/issues/4138

Figure 3.14: Selected bugs in Z3 and CVC4 found by YinYang.

```
(declare-fun a () String)
(declare-fun b () String)
(declare-fun d () String)
(declare-fun e () String)
(declare-fun f () Int)
(declare-fun g () String)
(declare-fun h () String)
(assert (or (not (= (str.replace
"B" (str.at "A" f) "") "B"))
(not (= (str.replace "B"
(str.replace "B" g "") "")
(str.at (str.replace
(str.replace a d "") "C" "")
(str.indexof "B" (str.replace
(str.replace a d "")
"C" "") 0))))))
(assert
    (= a (str.++ (str.++ d "C") g)))
(assert (= b (str.++ e g)))
(check-sat)
```

(a) Soundness bug in CVC4 reporting `sat` on this unsatisfiable QF_SLIA formula.

https://github.com/CVC4/CVC4/issues/3203

```
(declare-fun a () Real)
(declare-fun b () Real)
(declare-fun c () Real)
(declare-fun d () Real)
(declare-fun i () Real)
(declare-fun e () Real)
(declare-fun ep () Real)
(declare-fun f () Real)
(declare-fun j () Real)
(declare-fun g () Real)
(assert (or (not (exists ((h Real))
(=> (and (= 0.0 (/ b j)) (< 0.0 e))
(=> (= 0.0 i) (= (= (<= 0.0 h) (<=
h ep))(= 1.0 2.0))))))(not (exists
((h Real)) (=> (<= 0.0 (/ a h)) (= 0
(/ c e)))))))
(assert (= c (/ c g) g 0))
(assert (= ep (/ d f)))
(check-sat)
```

(b) Crash Bug in Z3: This NRA formula triggers a segmentation fault in Z3.

https://github.com/Z3Prover/z3/issues/2449

FIGURE 3.15: Ctd. Selected bugs in Z3 and CVC4 found by YinYang.

solvers are the same, but incorrect. The root cause of the bug is a missed corner case in the `str.to.int` reduction function for an empty string. The bug was labeled as *major* in the CVC4 bug tracker.

FIGURE 3.14E shows a soundness bug in Z3's QF_SLIA logic. According to Z3's main developer, this bug exposed and an incomplete axiomatization of the string to int conversion function `str.to_int`. Furthermore, he said that this bug inspired additional rewrite opportunities in Z3.

FIGURE 3.14F shows an invalid model bug in Z3's QF_SLIA logic. A buggy rewrite rule has caused this bug. The bug also led to an extensive discussion on mining rewrite rules by fuzzing on GitHub. Z3's main developer explained how he could infer a missing rewrite rule from the bug: From the term `(str.substr a 1 (- (str.len a))`, which occurs in the bug trigger, he inferred the new rewrite rule `(str.substr x y z) -> "" if z <= 0`. The function `(str.substr s i n)` evaluates to the longest (unscattered) substring of s of length at most `n` starting at position i. Clearly if `n` is negative, then the result is the empty string.

FIGURE 3.15A shows an unsatisfiable formula in the QF_SLIA logic. CVC4 incorrectly reported `sat` on this formula and gave an incorrect model.

The root cause for this bug is an unsound formula simplification of CVC4. The bug is labeled *major* by the CVC4 developers. They rewrote the simplification strategy to fix the bug.

FIGURE 3.15B shows a formula in quantified real arithmetic (NRA). Z3 crashed when solving this formula with the following message:

```
  Failed to verify: m_util.is_numeral(rhs, _k)
[2] 25133 segmentation fault (core dumped)
```

According to the bug fix of the developer, the root cause for this crash was an error in the rewriting strategy for the comparison operators <= and >=. According to the bug-fixing commit, the developer rewrote the rewrite for *less than or equal to* (<=) and *greater than or equal to* (>=) terms in arithmetic rewriter to fix the bug.

3.7 DISCUSSION

We first discuss effectiveness and reception, relate Semantic Fusion to more recent approaches, and then outline limitations and future work.

EFFECTIVENESS Our extensive evaluation demonstrates that YinYang can find many bugs in state-of-the-art SMT solvers Z3 and CVC4 (RQ1) and its findings are significant (RQ2). Further evaluation shows that YinYang can improve code coverage (RQ3) and that Semantic Fusion is necessary for YinYang's effectiveness in bug finding (RQ4).

RECEPTION Our testing effort produced significant, high-quality results. We specifically focused on the default modes of arithmetic and string solvers. As most users invoke SMT solvers in default modes, the bug reports are particularly valuable. YinYang found 39 critical soundness bugs, which shows the effectiveness of Semantic Fusion. Our bugs in Z3 have been fixed in the recent release versions, which makes Z3 more reliable and robust. In the following, we give a few developer's comments on our bug reports:

> *"Thanks a lot for finding this, this is fixed in my latest PR."*
>
> https://github.com/CVC4/CVC4/issues/3203#issuecomment-523102312
>
> *"Another excellent find, thanks a lot. This is fixed in my latest PR."*
>
> https://github.com/CVC4/CVC4/issues/3217#issuecomment-524364360
>
> *"Thanks a lot, this is another great bug find. This is fixed in my latest PR."*

https://github.com/CVC4/CVC4/issues/3272#issuecomment-531003582

"This is another great find, thank you! The issue should now be fixed on master"

https://github.com/CVC4/CVC4/issues/3357#issuecomment-538719981

"This was a bug in a rewrite rule. The example also exposed some opportunities for better rewrites."

https://github.com/Z3Prover/z3/issues/3926#issuecomment-613140447

"exposed: incomplete axiomatization of stoi; more opportunities for rewriting"

https://github.com/Z3Prover/z3/issues/4153#issuecomment-621317036

SEMANTIC FUSION, OPFUZZ, AND OTHER MORE RECENT APPROACHES Since the first publication of Semantic Fusion at PLDI 2020, researchers proposed three more approaches on SMT solver testing, worth discussing in this paragraph. One of them is STORM [32], a mutational fuzzer based on metamorphic testing. STORM generates mutants in a three-phase process of seed fragmentation, formula generation, and instance generation. STORM found 27 bugs in Z3, however none in CVC4. Another approach is BanditFuzz [58], a reinforcement learning-based fuzzer. Similar to StringFuzz, BanditFuzz's main focus is on performance issues in SMT solvers and less on correctness bugs. The authors have identified inconsistent results for 1600 syntactically different bug triggers on the four SMT solvers Z3, CVC4, MathSAT [42], Colibri, and 100 bug triggers in z3str3. However, the number of unique bugs in Z3 remains unclear as the authors did not reduce and report the bug triggers to filter out the duplicates.

OpFuzz [2], another follow-up work, has found several hundreds of bugs in the SMT solvers Z3 and CVC4. The key idea behind OpFuzz is to mutate SMT-LIB operators in a type-aware manner and then use differential testing to cross-check the solvers. Comparing our approach against OpFuzz is challenging. In the following, we can hence only give an approximation. OpFuzz's fuzzing campaign lasted for a year while YinYang's fuzzing campaign lasted for six months plus an additional period of less extensive testing. The fuzzing campaign with OpFuzz has focused on various logics and configurations of the solvers while our approach focused solely on (non-)linear arithmetic and string logics and solver default modes. Comparing the soundness bug findings for the logics which both YinYang and OpFuzz tested, OpFuzz found 81 soundness bugs in Z3 and 10 soundness bugs in CVC4 (all in default mode). Provided that OpFuzz's campaign was almost twice as long and the used seeds were richer (*e.g.*, the Z3 and CVC4 regression test suites were used containing non-standard SMT-LIB features such as tactics). YinYang is competitive although OpFuzz's absolute results are stronger than YinYang findings: 31 soundness bugs in Z3 and 5 for CVC4. While both approaches are empirically effective, the use cases for OpFuzz and YinYang are complementary. As OpFuzz is based on differential testing it needs at least two solvers implementing the tested features. Hence, testing new theories or language extensions available in a single solver is usually only feasible with Semantic Fusion, not with OpFuzz. A recent example is Z3's regex solver [59] which supports regexes with free variables. This feature is currently unsupported by CVC4. As a consequence, testing the regex solver with OpFuzz is often ineffective as CVC4 rejects mutants with free variables in regexes. Another example is (quantified) non-linear arithmetic for which Z3 has a very strong solver. CVC4, however, often returns `unknown` or times out on mutants with nonlinear arithmetic mutated by OpFuzz. Hence, OpFuzz is unable to test Z3's implementation of the non-linear arithmetic on such mutants. Furthermore, although rare, sometimes the two differentially tested solvers are both wrong, in which case OpFuzz would be unable to detect the bug. On the other hand, OpFuzz can be more easily extended to theories for which other strong solvers exist, as specifying type-aware operator mutations is simpler than designing fusion functions. Our work found 62 confirmed bugs in modern, mature, and widely-used SMT solvers, significantly more than any prior work on SMT solver testing before it. It is the first work to demonstrate that SMT solvers are much less stable than previously thought and inspired the follow-up research. Semantic Fusion is general and can find bugs in various logics while much prior work only focuses on the string logic.

LIMITATIONS AND FUTURE WORK While Semantic Fusion has demonstrated to be effective for SMT solver testing, it does also come with some limitations. First, Semantic Fusion relies on the given seed formulas. Second, one needs to manually design the fusion and inversion functions; devising variable fusion and variable inversion functions by hand can be difficult. It would be interesting future work to explore the automatic construction of variable fusion and inversion functions.

3.8 RELATED WORK

We discuss three strands of related work: (1) SMT solver validation, (2) validation of program analyzers, and (3) metamorphic testing.

SMT Solver Validation

PREVIOUS APPROACHES FuzzSMT [60] is the first effort targeting SMT solver validation. It is based on grammar-based blackbox fuzzing and differential testing. Brummayer and Biere evaluated FuzzSMT on the bit-vector logic and found 16 defects in five solvers, but no soundness bugs in Z3. BtorMBT [61] is a model-based testing tool for Boolector [62], an SMT solver for bit-vectors with arrays. It generates sequences of API calls to exploit the features of the solver. BtorMBT did not find bugs in any mature solvers. StringFuzz [56] focuses on performance issues in the string logic. It generates test cases by either mutating and transforming the benchmarks or generating random valid formulas. It found 3 performance and implementation bugs in z3str3 by differential testing. Different from these differential testing-based approaches, Semantic Fusion tackles the test oracle problem by construction, rather than cross-checking, making it capable of testing solver-specific features. Bugariu and Müller [63] proposed a formula synthesis approach to generating SMT formulas in the string logic with known satisfiability. It generates increasingly complex formulas via satisfiability-preserving transformations. Several bugs in Z3 were reported, while no reported bugs in CVC4. We instead fuse two formulas and preserve the satisfiability via fusion/inversion functions.

Validation of Program Analyzers

With program analyzers becoming increasingly practical and adopted, it is critical to ensure their reliability [48]. Several efforts explored this problem, targeting software model checkers, symbolic execution engines, and various static analyzers. Both Zhang *et al.* [64] and Klinger *et al.* [65] developed approaches to testing software model checkers — the approach by Zhang *et al.* [64] is based on reachability queries, while the approach by Klinger *et al.* [65] is based on differential testing. Kapus *et al.* [66] used random program generation and differential testing to find bugs in symbolic execution engines. Wu *et al.* [67] found bugs in alias analyses via cross-checking with dynamic aliasing information. Bugariu *et al.* [68] proposed an approach for finding soundness and precision bugs in numerical abstract domains. Qiu *et al.* [69] and Pauck *et al.* [70] reported experiences in testing and finding defects in analyzers for Android apps. Many of these program analyzers, such as software model checkers, symbolic execution engines, and program verifiers, critically rely on SMT solvers. Thus, our work also indirectly improves the reliability of program analyzers.

Metamorphic Testing

The test oracle problem is a longstanding challenge in software testing. Metamorphic testing is a general approach to this problem [71]. Its key idea is to leverage existing tests to construct additional ones with expected results via certain metamorphic relations. For example, the technique of equivalence modulo inputs (EMI) [72] is a notable instance of metamorphic testing for compilers. It constructs equivalent test programs for a seed program with respect to a given input by strategically mutating the seed program. To date, the general EMI-based approach and its variants [73, 74] have found more than 1,600 bugs in GCC and Clang/LLVM. EMI and metamorphic testing, in general, were also adapted to test shader compilers [75, 76]. The Semantic Fusion methodology introduced in this paper is also an instance of metamorphic testing — it generates new test formulas by fusing two existing test cases and preserving their oracle. Semantic Fusion is a new and highly generic metamorphic testing approach that we successfully applied to SMT solver testing.

4 TYPE-AWARE OPERATOR MUTATION

Semantic Fusion has demonstrated that SMT solvers are clearly less reliable than previously presumed. Yet it is unclear whether SMT solvers have now reached a strong level of maturity or whether many critical bugs remain. This chapter proposes Type-Aware Operator Mutation, a simple but unusually effective finding 1,254 unique bugs in Z3 and CVC4.

4.1 MOTIVATION

Satisfiability Modulo Theory (SMT) solvers are important tools for many programming language advances and applications. Incorrect results from SMT solvers can invalidate the results of these tools, which can be disastrous in safety-critical domains. Hence, the SMT community has undertaken great efforts to make SMT solvers reliable. Examples include the standardized input/output file formats for SMT solvers, semi-formal logic/theory specifications, extensive benchmark repositories, and yearly-held SMT solver competitions. To date, there are several mature SMT solvers, among which Z3 and CVC4 are the most prominent ones. Both Z3 and CVC4 are very stable and reliable. In Z3, there have been fewer than 150 reported soundness bugs in more than three years, while fewer than 50 in CVC4 in more than 8 years.[1] Despite this, SMT solvers are complex pieces of software and inevitably still have latent bugs. Various automated testing approaches [56, 60, 63] were devised for finding bugs in SMT solvers. However, nearly all SMT solver soundness bugs have still been exposed by applications, not by these techniques. This has only begun to change with the recently proposed works Semantic Fusion [1] and STORM [32]. Both exposed several soundness bugs in Z3, while Semantic Fusion additionally exposed some soundness bugs in CVC4. Yet, it is unclear whether SMT solvers have now reached a strong level of maturity or whether many critical bugs remain.

TYPE-AWARE OPERATOR MUTATION To answer these questions, we introduce Type-Aware Operator Mutation, a simple, yet unusually effective

[1] Data recorded before any SMT fuzzing campaigns: July 2010 to October 2019 for CVC4; April 2015 to October 2019 for Z3.

```
; \phi
(assert (forall ((a Int))
        (exists ((b Int))
        ( distinct  (* 2 b) a))))
(check-sat)
```

```
; \phi_{test}
(assert (forall ((a Int))
        (exists ((b Int))
        ( =  (* 2 b) a))))
(check-sat)
```

Figure 4.1: Type-aware operator mutation illustrated. We mutate the distinct operator in φ to the equals operator (see φ_{test}). Formula φ_{test} triggers a soundness bug in Z3 which reports `sat` on this unsatisfiable formula. https://github.com/Z3Prover/z3/issues/3973

approach for stress-testing SMT solvers. Its key idea is to mutate operators by other operators of conforming types within SMT formulas. Figure 4.1 illustrates type-aware operator mutation on an example formula. We replace the "`distinct`" in φ by an operator of conforming type, *e.g.*, the equals operator "=" to obtain formula φ_{test}. We then differentially test SMT solvers with φ_{test} as input and observe their results. If the results differ, *e.g.*, one SMT solver returns `sat` while the other returns `unsat`, we have found a soundness bug in either of the tested solvers. Formula φ_{test} inuitively reads as: "every integer number a is even". It is clearly unsatisfiable as b cannot exist whenever a is odd. In fact, while CVC4 correctly returns `unsat` on φ_{test}, Z3 incorrectly reports `sat` on φ_{test}. Thus, φ_{test} has triggered a soundness bug in Z3 which was promptly fixed by Z3's main developer.

BUG HUNTING WITH OPFUZZ We have engineered OpFuzz, a practical realization of Type-Aware Operator Mutation. OpFuzz is unusually effective. During our bug-hunting campaign from September 2019 to September 2020, we found and reported 1,254 bugs in Z3 and CVC4 issue trackers, among which 963 were confirmed by the developers and 917 were already fixed. We have found bugs across various logics such as (non-)linear integer and real arithmetic, uninterpreted functions, bit-vectors, strings, sets, sequences, arrays, floating-point, and combinations of these logics. Among these, most of the bugs (575) were found in the default modes of the solvers, *i.e.*, without additionally supplied options. This underpins the importance of our findings. We have found many high-quality soundness bugs in Z3 and notably also in CVC4, which has been proven to be a very robust SMT solver by previous work. The root causes of the bugs that we found are often complex and, sometimes require the developers to perform major code changes to fix the underlying issues. The developers of Z3 and CVC4

Approach	Bugs in Z3		Bugs in CVC4	
	soundness	all	soundness	all
StringFuzz [56]	0 (0)	1 (0)	-	-
BanditFuzz [58]	≥ 1 (0)	≥ 1 (0)	-	-
Bugariu and Müller [63]	3 (1)	5 (3)	0 (0)	0 (0)
YinYang [2]	25 (24)	39 (36)	5 (5)	9 (8)
STORM [32]	21 (17)	27 (21)	0 (0)	0 (0)
OpFuzz	**193** (128)	**674** (438)	**33** (14)	**289** (134)

FIGURE 4.2: Comparison of bugs found by OpFuzz against other SMT solver testing approaches. Snapshot at October 15, 2020. In parentheses: confirmed bugs in the default modes of the solvers. Performance issues are excluded from this comparison.

greatly appreciated our bug-finding effort with comments like "Great find!", "Thanks a lot for the bug report!" or labeling our bug reports as "major".

COMPARISON OF OPFUZZ WITH RECENT SMT FUZZERS Compared to other fuzzers of other fuzzing campaigns, OpFuzz found orders of magnitude more critical bugs in Z3 and CVC4. Figure 4.2 compares OpFuzz with recent SMT solver fuzzer from the last two years. We use the term bug trigger to refer to a formula that triggers a bug in an SMT solver and the term bug to refer to a single unique bug in an SMT solver. Note, bug triggers can be caused by the same underlying bug. All bug counts mentioned in the comparison and refer to unique bugs. We have not considered older approaches and defer to the related work section (Section 4.6).

Prior approaches can be roughly separated into two categories: generators (StringFuzz [56], Bugariu and Müller's approach [63], BanditFuzz [58]) and mutators (StringFuzz, YinYang [1], STORM [32]). StringFuzz is a string formula generator that also comes with a mutator. It mainly targets performance issues in z3str3 [77], an alternate string solver in Z3. StringFuzz can find correctness bugs as a by-product; the paper mentioned one. Bugariu and Müller's approach is a formula synthesizer for string logic generating formulas that are by construction (un)satisfiable. They found 5 bugs in Z3 in total with 3 soundness bugs, but none in CVC4. Recently, BanditFuzz, a reinforcement learning-based fuzzer has been proposed. Similar to StringFuzz, BanditFuzz's main focus is on performance issues in SMT solvers and less on correctness bugs. The authors have identified inconsistent results for 1,600 syntactically different bug triggers on the four SMT solvers Z3, CVC4, MathSAT [42], Colibri, and 100 bug triggers in z3str3. However, the number of unique bugs in Z3 remains unclear as the authors did not reduce and report the bug triggers to filter out the duplicates. Among the mutation-based fuzzers, YinYang is an approach to stress-test SMT solvers by fabricating fused formula pairs that are by construction either (un)satisfiable. YinYang found 39 bugs in Z3 and 9 in CVC4. Another recent approach is STORM which is based on a three-phase process of seed fragmentation, formula generation, and instance generation. STORM has found 27 bugs in Z3 with 21 being soundness bugs, but none in CVC4. As Figure 4.2 illustrates, our realization OpFuzz of type-aware operator mutation compares favorably against all existing approaches by a significant margin — OpFuzz found substantially more bugs in both Z3 and CVC4 in terms of all bugs, the soundness bugs in Z3 and CVC4, and bugs for the default modes of the solvers. Existing approaches also extensively tested Z3 and CVC4, and have missed the bugs found by OpFuzz.

4.2 ILLUSTRATIVE EXAMPLES

We first examine three exemplary bugs that were found by our technique. Consider the formula in Figure 4.3a on which CVC4 returns the following model: $a = -\frac{3}{2}$ and $b = -\frac{1}{2}$. This model is invalid as $a \cdot b \neq 1$. Mutating the equals operator = to the greater operator > hides this bug (see Figure 4.3b). As another example, consider the formula in Figure 4.3c. CVC4 gives an invalid model on this formula by setting $f = g = \mathit{false}$. Furthermore, CVC4 crashes on the formula in Figure 4.3e. Again in both cases, the bug disappears with a single operator change (see Figure 4.3b and Figure 4.3d). All illustrated cases show that operators play an important role in triggering SMT solver bugs. This inspired our technique, Type-Aware Operator Mutation, that is to stress-test SMT solvers via mutating operators and use the so mutated formulas for stress-testing SMT solvers.

However, substituting an operator with another arbitrary operator may not always yield a syntactically correct formula. As an example, consider Figure 4.4a that presents a syntactically correct seed formula. By substituting the first operator greater than operator > to $*$, the formula becomes syntactically incorrect (see Figure4.4b). This is because the `assert` statement expects a boolean expression, while $*$ returns a real. The formula of Figure 4.4b is of little value to testing an SMT solver's decision procedures

```
(set-logic NRA)
(declare-fun a () Real)
(declare-fun b () Real)
(assert (= (* a b) 1))
(check-sat) (get-model)
```

(a) Invalid model bug in CVC4. https://github.com/CVC4/CVC4/issues/3407

```
(set-logic NRA)
(declare-fun a () Real)
(declare-fun b () Real)
(assert (> (* a b) 1))
(check-sat) (get-model)
```

(b) Mutating the equals operator in Figure 4.3a to a greater operator makes the bug disappear.

```
(declare-fun f (Int) Bool)
(declare-fun g (Int) Bool)
(assert (distinct f g))
(check-sat)
(get-model)
```

(c) Invalid model bug in CVC4. https://github.com/CVC4/CVC4/issues/3527

```
(declare-fun f (Int) Bool)
(declare-fun g (Int) Bool)
(assert (= f g))
(check-sat)
(get-model)
```

(d) Mutating the equals operator in Figure 4.3c to a distinct operator makes the bug disappear.

```
(declare-fun x () Real)
(assert (distinct x (sin 4.0)))
(check-sat)
```

(e) CVC4 crashes on this formula. https://github.com/CVC4/CVC4/issues/3614

```
(declare-fun x () Real)
(assert (>= x (sin 4.0)))
(check-sat)
```

(f) Mutating the distinct operator in Figure 4.3e to a greater than operator makes the bug disappear.

Figure 4.3: Left column: bug-triggering formulas in SMT-LIB format. Right column: formulas that were transformed from the corresponding bug-triggering formulas by a single operator change.

since the solvers would reject such formulas already at a preprocessing stage. Hence, we have to consider the operator types for the substitutions, *i.e.*, avoid substituting an operator returning a boolean value, such as `=`, by an operator returning a real, such as `*`; neither should we substitute an operator with a single argument, like `not`, by an operator of two or more arguments, such as `=`. Instead, we mutate the operators in a *type-aware* fashion. Consider the first `>` of the formula in Figure 4.4a. It takes an arbitrary number of numeral arguments and returns a boolean. Candidates for its substitution are `<=`, `>=`, `<`, `=`, and `distinct`, all of which have a conforming type, *i.e.*, read more than one numerals and return a boolean. Therefore, we can safely substitute `>` of the formula in Figure 4.4a with

```
(declare-fun a () Real)
(assert ( > (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(a) Original formula

```
(declare-fun a () Real)
(assert ( * (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(b) Syntactically incorrect mutant.

```
(declare-fun a () Real)
(assert ( = (/ (* 2 a) a) (* a a) 1))
(check-sat)
```

(c) Syntactically correct mutant.

```
(declare-fun a () Real)
(assert (= (/ (* 2 a) a) ( / a a) 1))
(check-sat)
```

(d) Bug triggering mutant.

Figure 4.4: Motivating examples for Type-Aware Operator Mutation.

a random candidate, *e.g.*, `=`. As a result, we obtain the mutant formula in Figure 4.4c. This formula is syntactically correct and can successfully pass the preprocessing phase of the SMT solvers. We call such mutations *type-aware operator mutations*. As we have the guarantee that the mutant is a type-correct formula, we can do iterative type-aware operator mutations. Given the mutant formula in Figure 4.4c, we further substitute the second occurrence of `*` with `/` safely. This yields the formula in Figure 4.4d which triggered a soundness bug in Z3. Division by zero terms are specified in the Real and Int theories of the SMT-LIB as the uninterpreted terms, meaning that for a term `(/ t 0)` and arbitrary value `v`, the equation `(= v (/ t 0))` is satisfiable. In fact, we can set $a = 0$ to realize a model for the formula in Figure 4.4d, *i.e.*, let the division by zero terms be 1 to satisfy the assert. Hence, the formula in Figure 4.4d is satisfiable. However, Z3 incorrectly reports `unsat` on it.

4.3 TYPE-AWARE OPERATOR MUTATION

In this section, we formally introduce Type-Aware Operator Mutation and propose OpFuzz, a fuzzer for stress-testing SMT solvers.

BACKGROUND For a formula φ, we define $F(\varphi)$ to be φ's set of (enumerated) function occurrences. For example, for $\varphi =$ `(+ (* 1 1) (- 2 (* 5 2)))`, we have: $F(\varphi) = \{\texttt{+}_1, \texttt{*}_1, \texttt{-}_1, \texttt{*}_2\}$. Formula $\varphi[f_1/f_2]$ describes the substitution of function f_2 by f_1 in φ. Expressions and functions are typed. For example, `1` is of type *Int*, `1.0` is of type *Real*, `"foo"` is of type *String*. Similarly, functions also have types. We denote the type of a function f by $f : A \to B$ where A is the type of its arguments and B its return type. We

Function types	**Function Symbols**
$\Gamma, A <: \top \vdash A \times \cdots \times A \rightarrow \mathit{Bool}$	`=, distinct`
$\Gamma \vdash \mathit{Quantifier} \times \mathit{Bool} \rightarrow \mathit{Bool}$	`exists, forall`
$\Gamma \vdash \mathit{Bool} \times \cdots \times \mathit{Bool} \rightarrow \mathit{Bool}$	`and, or, =>`
$\Gamma, \mathit{Int} <: \mathit{Real} \vdash \mathit{Real} \times \cdots \times \mathit{Real} \rightarrow \mathit{Bool}$	`<=, >=, <, >`
$\Gamma, \mathit{Int} <: \mathit{Real} \vdash \mathit{Real} \times \cdots \times \mathit{Real} \rightarrow \mathit{Real}$	`+, -, *, /`
$\Gamma \vdash \mathit{Int} \times \cdots \times \mathit{Int} \rightarrow \mathit{Int}$	`div`
$\Gamma \vdash \mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}$	`mod`

FIGURE 4.5: SMT function symbols categorized by their type.

use Γ to denote the static typing environment of the SMT-LIB language. For example, we write $\Gamma \vdash \mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}$ for the type of function `mod` and $\Gamma \vdash \mathit{Int} \times \cdots \times \mathit{Int} \rightarrow \mathit{Int}$ for the function `div`. $\mathit{Int} \times \cdots \times \mathit{Int}$ means function `div` accepts more than one argument with type *Int*. Figure 4.5 shows selected functions and their types. We emphasize that our theory is not restricted to the functions used in Figure 4.5. It can be extended to a richer set of functions and types according to the SMT-LIB standard. Similar to other programming languages with types, we can define a subtyping relation for the SMT-LIB language. We now formalize a fragment of the SMT-LIB's type system. We define type *Int* to be a subtype of *Real*, *i.e.*, $\Gamma \vdash \mathit{Int} <: \mathit{Real}$. Let A be an arbitrary type, then we define type $A \times A$ to be a subtype of $A \times \cdots \times A$, *i.e.*, $\Gamma, A <: \top \vdash A \times A <: A \times \cdots \times A$. For two functions $f_1 : A_1 \rightarrow B_1$ and $f_2 : A_2 \rightarrow B_2$ with $A_1 <: A_2$ and $B_2 <: B_1$:

$$\frac{A_1 <: A_2 \quad B_2 <: B_1}{f_2 : A_2 \rightarrow B_2 <: f_1 : A_1 \rightarrow B_1}$$

For example, consider the function `div` of type $\mathit{Int} \times \cdots \times \mathit{Int} \rightarrow \mathit{Int}$ and function `mod` of type $\mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}$ from Figure 4.5. We can hence conclude that `div`'s type is a subtype of `mod`'s type:

$$\frac{\Gamma \vdash \mathit{Int} \times \mathit{Int} <: \mathit{Int} \times \cdots \times \mathit{Int} \quad \Gamma \vdash \mathit{Int} <: \mathit{Int}}{\Gamma \vdash \mathit{Int} \times \cdots \times \mathit{Int} \rightarrow \mathit{Int} <: \mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}}$$

We call φ *well-typed* if it complies with the rules of SMT-LIB's type system.

Having provided basic background, we present Type-Aware Operator Mutation, the key concept of this chapter. We first introduce type-aware operator mutations and then show that type-aware operator mutants realize well-typed SMT-LIB programs.

Definition 4.3.1 (Type-aware operator mutation)**.** Let φ be an SMT formula and let $f_1 : t_1$ and $f_2 : t_2$ be two of its functions. We say formula $\varphi' = \varphi[f_2/f_1]$ is a **type-aware operator mutant** of φ if $t_2 <: t_1$. Transforming φ to $\varphi[f_2/f_1]$ is called **type-aware operator mutation**.

Proposition 4.3.1. *Type-aware operator mutants are well-typed.*

Proof. Let φ be a well-typed SMT formula and let φ' be a type-aware operator mutant of φ. According to Definition 4.3.1 we know that $\varphi' = \varphi[f_2/f_1]$ where $f_1 : t_1$ and $f_2 : t_2$ are two of φ's functions. By Definition 4.3.1, we also know $t_2 <: t_1$. This implies that all arguments of f_1 are also accepted by f_2 and all values returned by f_2 could be produced by f_1. Thus, f_2 accepts all the inputs provided by φ', and formula φ' accepts all the outputs of f_2. Therefore formula φ' is well-typed. □

Example 4.3.1. Consider the following formula:

$$\varphi = \texttt{(assert (= (mod 1 1) 1)}$$

with $F(\varphi) = \{$`=`,`mod`$\}$. We randomly pick function `mod` from F and substitute it with a function that has its subtype, *e.g.*, the function `div`. We get the following type-aware operator mutant $\varphi' =$ `(assert (= (div 1 1) 1)`. As Proposition 4.3.1 shows, φ' is guaranteed to be well-typed Thus, we can use formula φ' for testing SMT solvers.

OPFUZZ We implemented OpFuzz, a type-aware operator mutation-based fuzzer, for stress-testing SMT solvers. OpFuzz leverages type-aware operator mutation to generate test inputs and validates the results of the SMT solvers via differential testing, *i.e.*, by comparing the results of two or more SMT solvers and reporting their inconsistencies. Algorithm 3 presents the main process of OpFuzz. OpFuzz takes a set of seed formulas `Seeds`, two SMT solvers S_1, S_2 and parameter n as its input. OpFuzz collects bug triggers in the set *triggers* which is initialized to the empty set. The main process runs inside a while loop until an interrupt is detected, *e.g.*, by the user or by a time or memory limit that is reached. We first choose a random formula φ from the set of formulas `Seeds` for initialization. In the while loop, we then perform a type-aware operator mutation on φ realized by the `type_aware_op_mutate` function. In the `type_aware_op_mutate` function

Algorithm 3 OpFuzz's main process.

```
procedure OpFuzz(Seeds, $S_1$, $S_2$, $n$)
    $triggers \leftarrow \emptyset$
    while true do
        $\varphi \leftarrow random.choice($Seeds$)$
        for $i = 1$ to $n$ do
            $\varphi' \leftarrow$ TYPE_AWARE_OP_MUTATE($\varphi$)
            if $\neg$VALIDATE($\varphi', S_1, S_2$) then
                $triggers \leftarrow triggers \cup \{\varphi'\}$
            $\varphi \leftarrow \varphi'$
        if Interruption then
            break
    return triggers
```

(Algorithm 4), we first randomly pick a function f_1 from the set of functions in φ. Then, we randomly choose a function f_2 from the set of f_1's subtypes. The subtype function is realized based on Figure 4.5. After we obtained $\varphi' = \varphi[f_2/f_1]$ by type-aware mutation on φ, we call the function `validate`. It cross-checks two SMT solvers S_1 and S_2 via differential testing on the input formula φ'. First, it checks whether either of the solvers has produced an error on processing φ', e.g., the SMT solver did not terminate successfully, throwing out an error message. We distinguish two cases: either φ' triggered an assertion violation or segmentation fault (crash), or a model validation error that occurs for solvers with model validation enabled (invalid model). In both cases, the function returns *false*. Otherwise, it checks whether the results of the solvers are different, and returns *false* if so, else `validate` returns *true* indicating that φ' has not exposed a bug trigger in either of the solvers S_1 and S_2. OpFuzz realizes an n-times repeated type-aware operator mutation on every seed formula. For the parameter n, a value within 200 and 400 has worked well in practice.

OpFuzz is very light-weight. We realized OpFuzz in a total of only 212 lines of Python 3.7 code. OpFuzz can be run in parallel mode, which can significantly increase its throughput. Users can customize OpFuzz's command-line interface to test specific solvers and/or configurations. OpFuzz can be used with any SMT solver that takes SMT-LIB v2.6 files as its input. We implemented the mutations *w.r.t.* the function symbols in Figure 4.5.

Algorithm 4 Function realizing type-aware operator mutations and function for differentially testing of the SMT solvers S_1 and S_2.

```
function TYPE_AWARE_OP_MUTATE($\varphi$)
    $f_1 \leftarrow random.choice(F(\varphi))$
    $f_2 \leftarrow random.choice(subtypes(f_1))$
    return $\varphi[f_2/f_1]$

function VALIDATE($\varphi', S_1, S_2$)
    if $S_1(\varphi') = \text{error} \;\vee\; S_2(\varphi') = \text{error}$ then
        return $false$
    if $\vee\ S_1(\varphi') \neq S_2(\varphi')$ then
        return $false$
    return $true$
```

4.4 EMPIRICAL EVALUATION

This section details our extensive evaluation with OpFuzz demonstrating the practical effectiveness of Type-Aware Operator Mutation for testing SMT solvers. Between September 2019 and September 2020, we were running OpFuzz to stress-test the SMT solvers Z3 [38] and CVC4 [39]. We have chosen Z3 and CVC4, since they (1) both are popular and widely used in academia and industry, (2) support a rich set of logics, and (3) adopt an open-source development model. During our testing period, we have filed numerous bugs on the issue trackers of Z3 and CVC4. This section describes the outcome of our fuzzing campaign and efforts.

RESULT SUMMARY AND HIGHLIGHTS OpFuzz is unusually effective.

- *Many confirmed bugs:* In one year, we have reported 1,254 bugs, and 963 unique bugs in Z3 and CVC4 have been confirmed by the developers.
- *Many soundness bugs:* Among these, there were 226 soundness bugs in Z3 and CVC4. Most notably, we have found 33 in CVC4.
- *Most logics affected:* Our bug findings affect most SMT-LIB logics including strings, (non-)linear integer and real arithmetic, bit-vectors, uninterpreted functions, floating points, arrays, sets, sequences, horn, and combinations thereof.

- *Most bugs in default modes:* 575 out of our confirmed 963 bugs are in the default modes of the solvers.

4.4.1 *Evaluation Setup*

HARDWARE SETUP AND TEST SEEDS We have run OpFuzz on an AMD Ryzen Threadripper 2990WX processor with 32 cores and 32GB RAM on an Ubuntu 18.04 64-bit. As test seeds, we have mainly used the SMT-LIB benchmarks [47]. We chose the SMT-LIB benchmarks as our test seeds since they make the largest collection of SMT formulas in the SMT-LIB 2.6 language. These SMT-LIB benchmarks are also used in the SMTComp, the annual SMT solver competition. Therefore, they are unlikely to trigger bugs in Z3 and CVC4 since they have already been run on them. In addition to the SMT-LIB benchmarks, we used the regression test suites of Z3 [78] and CVC4 [79]. We show the seed formula counts categorized by logic and solving mode in Appendix A.2. We treated all seed files equally during fuzzing. The effort spent on testing for a specific logic is therefore proportional to the number of its seed files within the overall seed set. Consequently, logics with a high seed count get tested more frequently as compared to others with a lower seed count. We regularly ran Z3 and CVC4 on all seed files and excluded bug-triggering seeds but have very rarely encountered any bug-triggering seed formulas.

TESTED OPTIONS AND FEATURES We mainly focused our testing efforts on the default modes of the solvers. For CVC4, this includes enabling the options `--produce-models`, `--incremental` and `--strings-exp` as needed to support all test seed formulas. To detect invalid model bugs, we have supplied `--check-models` to CVC4 and `model.validate=true` to Z3. We consider these to be part of the default mode for the two solvers Z3 and CVC4 if apart from these necessary options, no other options or tactics were used. Besides the default modes of Z3 and CVC4, we have considered many frequently used options and solver modes for Z3 and CVC4 of which we only detail a subset here. For Z3, we have stress-tested several tactics and several arithmetic solvers including `smt.arith_solver=x` with $x \in \{1, \cdots, 6\}$. We have also tested, among others, the string solver z3str3 by supplying `smt.string_solver=z3str3`. In CVC4 we have tested, among many other options, syntax-guided synthesis procedure [80] by specifying `--sygus-inference` and higher-order reasoning for UF `--uf-ho`.

BUG TYPES We have encountered many different kinds of bugs and issues while testing SMT solvers. We distinguish them by the following categories with two SMT solvers S_1 and S_2.

- *Soundness bug*: Formula φ triggers a soundness bug if solvers S_1 and S_2 both do not crash and give different results on φ.
- *Invalid model bug*: Formula φ triggers an invalid model bug if the model returned by the solver does not satisfy φ.
- *Crash bug*: Formula φ triggers a crash bug if the solver throws out an assertion violation or a segmentation fault while solving φ.

OpFuzz detects soundness bug triggers by comparing the standard outputs of the solvers. It detects invalid model bug triggers by internal errors when using the SMT solver's model validation configuration. A crash bug trigger is detected if a solver returns a non-zero exit and no timeout occurs.

BUG TRIGGER DE-DUPLICATION OpFuzz collects bug triggers that may stem from the same underlying bug. Hence, we de-duplicated the bug triggers after each fuzzing run to avoid duplicate bug reports on the GitHub issue trackers. Crash bugs are either assertion violations or segmentation faults. We de-duplicate assertion violations via the location information (file name and line number) printed on standard output/error. We de-duplicate segmentation faults by comparing their ASAN traces. For soundness and invalid model bugs, we used the following procedure. We first categorize the bug triggers by theory. We do this because bug triggers in different theories are likely to be unique bugs. Then, we select one bug trigger per theory at a time for reporting. If the bug was fixed, we checked the remaining bug-triggering formulas of the same theory. If either one of them still triggered a bug in the solver, we repeat this process until none of the remaining formulas triggers a bug anymore.

BUG REDUCTION If a bug trigger is selected in the trigger de-duplication, we reduce the bug-triggering formula to a small enough size for reporting. We use C-Reduce [54], a C code reduction tool that also works for the SMT-LIB language. We implemented a pretty printer to help with the bug reduction process, *e.g.*, when C-Reduce has converged to a still very large formula or hangs. The pretty-printer makes simple modifications to the abstract syntax tree of the formula, *e.g.*, flattens nestings of the same operator, removes additions and multiplications with neutral elements, and returns the modified formula in a human-readable format.

Status	Z3	CVC4	Total
Reported	915	339	1,254
Confirmed	674	289	963
Fixed	656	261	917
Duplicate	96	20	116
Won't fix	124	29	153

(a)

Type	Z3	CVC4	Total
Crash	339	209	548
Soundness	193	33	226
Invalid model	111	27	138
Others	31	20	51

(b)

#Options	Z3	CVC4	Total
default	442	133	575
1	138	86	224
2	51	31	82
3+	43	39	82

(c)

FIGURE 4.6: (a) Status of bugs found in Z3 and CVC4. (b) Bug types among the confirmed bugs. (c) # Solver options among the confirmed bugs.

4.4.2 *Evaluation Results*

Having defined the setup and bug types, we continue with the presentation of the evaluation results. The section is divided into three parts: (1) statistics on the bug findings by OpFuzz to assess its effectiveness, (2) coverage measurements of OpFuzz relative to the seed formulas (3) solver trace comparisons to gain further insights into the technique.

BUG FINDINGS Figure 4.6a shows the bug status counts. By "Reported", we refer to the unique bugs after bug trigger de-duplication that we posted on the GitHub issue trackers of the solvers; by "Confirmed", we refer to those posted bugs that were confirmed by the developers as unique bugs; by "Fixed", we refer to those posted bugs that were confirmed by the developers as unique bugs and addressed through at least one bug-fixing commit; by "Duplicate", we refer to those bugs posted on GitHub that have been identified by the developers as duplicate to another bug report of ours or to a previously existing bug report; by "Won't fix", we refer to those posted bugs that were rejected by the developers, due to misconfigurations.

We have reported a total of 1,254 bugs on Z3's and CVC4's respective issue trackers. Among these, 963 unique bugs were confirmed and 917 were fixed. Although we devoted equal testing effort to both solvers, we found more than twice as many bugs in Z3 as in CVC4. Previous approaches made similar observations [1].

Figure 4.6b shows the bug types. Among the bug types of the confirmed bugs, crash bugs were most frequent (548), followed by soundness bugs (226) and invalid model bugs (138). The type "Others" refers to all other unexpected behaviors in SMT solvers such as rejecting syntax-correct formulas, alarming invalid models when generating a valid model. The large

Logic	S	I	C	O	Total
QF_S	47	32	45	9	133
NRA	21	0	46	1	68
QF_NRA	16	10	20	8	54
QF_SLIA	16	8	20	0	44
QF_LIA	6	10	26	1	43
QF_NIA	17	4	16	2	39
UFLIA	4	6	16	0	26
QF_FP	1	10	11	0	22
Uncategorized	5	0	16	1	22
UF	3	1	18	0	22
QF_BV	8	4	9	0	21
LIA	6	0	13	0	19
QF_LRA	5	2	7	2	16
QF_UF	1	6	8	0	15
NIA	5	0	9	0	14
QF_UFLIA	1	6	6	0	13
LRA	4	1	8	0	13
QF_NIRA	3	5	1	2	11
Horn	4	2	5	0	11
QF_AX	2	1	7	0	10
ALIA	1	0	9	0	10
BV	3	0	6	0	9
NIRA	3	2	1	3	9
Set	1	0	5	0	6
UFIDL	2	0	2	0	4
QF_ABV	3	0	0	0	3
AUFNIRA	0	1	2	0	3
FP	2	0	1	0	3
Sequences	0	0	1	1	2
UFLRA	0	0	2	0	2
QF_ABVFP	0	0	2	0	2
ABV	1	0	0	0	1
UFNIA	1	0	0	0	1
QF_UFIDL	0	0	1	0	1
QF_LIRA	0	0	0	1	1
QF_UFNRA	1	0	0	0	1
Total	**193**	**111**	**339**	**31**	**674**

(a) Z3

Logic	S	I	C	O	Total
QF_NRA	3	4	20	4	31
NRA	1	2	15	4	22
Set	3	0	15	1	19
Uncategorized	2	2	14	0	18
QF_S	6	2	9	1	18
UFLIA	0	2	16	0	18
QF_LIA	3	0	13	0	16
UF	1	0	15	0	16
QF_BV	1	1	14	0	16
LIA	2	0	12	0	14
QF_LRA	1	0	6	3	10
BV	1	0	8	0	9
LRA	0	0	9	0	9
QF_UF	1	0	7	0	8
QF_FP	1	0	6	0	7
QF_NIA	2	0	4	0	6
QF_SLIA	1	2	1	2	6
NIA	1	0	3	1	5
QF_AX	0	0	5	0	5
QF_ABV	0	2	3	0	5
QF_UFLIA	1	2	1	0	4
QF_NIRA	1	0	0	3	4
QF_ABVFP	0	0	3	0	3
Sequences	0	1	2	0	3
QF_AUFLIA	0	3	0	0	3
QF_AUFBVLIA	0	2	1	0	3
ALRA	0	0	2	0	2
QF_UFIDL	0	0	1	0	1
AUFNIRA	0	1	0	0	1
NIRA	0	0	1	0	1
ALIA	0	0	0	1	1
QF_ALIA	1	0	0	0	1
UFBV	0	0	1	0	1
UFNIRA	0	0	1	0	1
QF_UFLRA	0	0	1	0	1
QF_UFNRA	0	1	0	0	1
Total	**33**	**27**	**209**	**20**	**289**

(b) CVC4

FIGURE 4.7: Logic distribution of the confirmed bugs: (S) soundness bugs, (I) invalid model bugs, (C) crash bugs, and (O) others. "Uncategorized" refers to bugs that could not be associated with a single logic.

majority (575 out of 963) of bugs found by OpFuzz were found in the default modes of the solvers, *i.e.*, no additional options were supplied, some were found with one or two additional options enabled, and clearly less bugs with more than three options enabled (see Figure 4.6c).

We also examined the distribution of logics among the confirmed bugs of Z3 and CVC4 (see Figure 4.7a and 4.7b). We observe that most soundness bugs in Z3 are in the string logics QF_S (133), QF_SLIA (44) and nonlinear logics NRA (20), QF_NRA (8). Notably, there are also a number of soundness bugs in bitvectors QF_BV (8) and linear real and integer arithmetic QF_LRA (4), QF_LIA (6), LIA (6). Similar to Z3, most soundness bugs in CVC4 are also in the string logic QF_S (6) and nonlinear arithmetic QF_NRA (3). Moreover, there are three soundness bugs in set logics.

CODE COVERAGE OF OPFUZZ'S MUTATIONS Code coverage is a reference for the sufficiency of software testing. This experiment aims to answer whether the mutants generated by OpFuzz can achieve higher coverage than the seed formulas. We randomly sampled 1000 formulas ($\texttt{Seeds}_{1000}$) from all formulas that we used for stress-testing SMT solvers. We instantiated OpFuzz with $n = 300$, run OpFuzz on the seeds $\texttt{Seeds}_{1000}$ and then measure the cumulative line/function/branch coverage over all formulas and runs.[2] For all coverage measurements, we used Gcov[3] from the GCC suite.

The results show that OpFuzz increases the code coverage upon $\texttt{Seeds}_{1000}$ (Figure 4.8). Z3 and CVC4 have over 436K LoC and 238k LoC respectively, so that 0.1% improvement already translates to hundreds of additionally covered lines. However, although noticeable, the coverage increments are not significant ($\leq 0.5\%$). A partial explanation is that decision procedures of Z3 and CVC4 are highly recursive. This leads to many calls of the same functions with different arguments. Hence the difference in line/function/branch coverage achieved by different formulas of the same theory, may not be as significant. This experiment also provides further evidence that standard coverage metrics (*e.g.*, statement and branch coverages), although useful, are insufficient for measuring the thoroughness of testing. Indeed, such small coverage increases led to 1,254 new bugs in two of the most mature, widely used SMT solvers.

EXECUTION TRACE COMPARISON Since code coverage could not thoroughly explain the effectiveness of OpFuzz, we also examine the internals

[2] This makes a total of 300k runs.
[3] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

	Z3			CVC4		
	lines	*functions*	*branches*	*lines*	*functions*	*branches*
$\texttt{Seeds}_{1000}$	33.2%	36.2%	13.7%	28.5%	47.1%	14.3%
OpFuzz	**33.5%**	**36.4%**	**13.8%**	**28.8%**	**47.4%**	**14.4%**

FIGURE 4.8: Line, function and branch coverage achieved by the baseline $\texttt{Seeds}_{1000}$ versus OpFuzz on Z3 and CVC4's respective source codes.

of the solvers by investigating the similarity of their execution traces upon type-aware operator mutations. *What is the relative similarity of the execution traces with respect to the seed?* In the following experiment, we approach this question. In Z3 and CVC4, we can obtain an execution trace by setting the flags `TRACE=True` and `--trace-theory` respectively. Before describing this experiment, we first show the format of Z3's and CVC4's respective traces via an example. Consider formula φ and its type-aware operator mutation φ_{mutant} (see Figure 4.9a and 4.9d). Figure 4.9c and 4.9e shows Z3's and CVC4's traces on solving φ respectively, Figure 4.9d and 4.9f show Z3's and CVC4's traces on solving φ_{mutant} respectively.

Having obtained an intuition of the execution traces, we now get to the actual experiment. Our aim is to measure the relative change in the execution traces of Z3 and CVC4. We therefore perform 40 mutation steps for every formula in $\texttt{Seeds}_{1000}$ and record the execution trace triggered in each step. To quantify the similarity of two traces t_1 and t_2, we compute a metric $sim(t_1, t_2)$ with

$$sim(t_1, t_2) = \frac{2 \cdot |LCS(t_1, t_2)|}{\#lines(t_1) + \#lines(t_2)}$$

where $LCS(t_1, t_2)$ corresponds to the longest common subsequence of t_1 and t_2; $\#lines(t_1)$ and $\#lines(t_2)$ are the number of lines in t_1 and t_2 respectively. As an example, re-consider Figure 4.9. The differing lines of φ's trace and φ_{mutant}'s trace are shaded. φ's Z3 trace and φ_{mutant}'s Z3 trace match in 10 out of 11 lines and therefore their similarity score is $\frac{10}{11}$. For the trace pair of CVC4, the number of longest common subsequence is of length 3 and hence the similarity of Z3's trace is $\frac{1}{2}$. To compute the longest common subsequence, we used the `difflib`[4] package from python's standard library. Note that type-aware operator mutation may rename the AST node identi-

[4] https://docs.python.org/3/library/difflib.html

```
;phi
(declare-fun a () Real)
(declare-fun b () Real)
(assert (< a 0))
(assert (< b 0))
(check-sat)
```

(a)

```
;phi_mutant
(declare-fun a () Real)
(declare-fun b () Real)
(assert (> a 0))
(assert (< b 0))
(check-sat)
```

(b)

```
[mk-app] #23 a
[mk-app] #24 Int
[attach-meaning] #24 arith 0
[mk-app] #25 to_real #24
[mk-app] #26 < #23 #25
[mk-app] #27 Real
[attach-meaning] #27 arith 0
[inst-discovered] theory-solving 0
arith# ; #25
[mk-app] #28 = #25 #27
[instance] 0 #28
[attach-enode] #28 0
```

(c)

```
[mk-app] #23 a
[mk-app] #24 Int
[attach-meaning] #24 arith 0
[mk-app] #25 to_real #24
[mk-app] #26 > #23 #25
[mk-app] #27 Real
[attach-meaning] #27 arith 0
[inst-discovered] theory-solving 0
arith# ; #25
[mk-app] #28 = #25 #27
[instance] 0 #28
[attach-enode] #28 0
```

(d)

```
TheoryEngine::assertFact
  (not (>= b 0.0)) (0 left)
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= a 0.0)), false)
TheoryEngine::assertFact
  ((not (>= b 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= b 0.0)), false)
Theory::get() =>
  (not (>= a 0.0))(1 left)
Theory::get() =>
  (not (>= b 0.0)) (0 left)
```

(e)

```
TheoryEngine::assertFact
  ((not (>= (* (- 1.0) a) 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= (* (- 1.0) a) 0.0)), false)
TheoryEngine::assertFact
  ((not (>= b 0.0)))
Theory<THEORY_ARITH>::assertFact[1]
  ((not (>= b 0.0)), false)
Theory::get() =>
  (not (>= (* (- 1.0) a) 0.0)) (1 left)
Theory::get() =>
  (not (>= b 0.0)) (0 left)
```

(f)

FIGURE 4.9: Left column: (a) seed formula φ (b) Z3 trace snippet of φ and (c) CVC4 trace snippet of φ. Right column: (d) type-aware operator mutant φ_{mutant} (e) Z3 trace snippet of φ_{mutant} (f) CVC4 trace snippet of φ_{mutant} of φ. Differences are shaded.

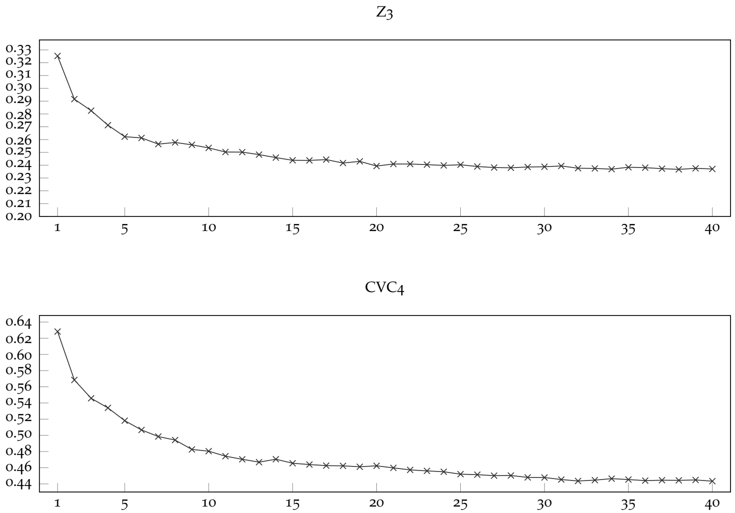


FIGURE 4.10: Average similarity of consecutively generated mutants (y-axis) per mutation step (x-axis). y-axis: represents the similarity between the mutant generated in the corresponding mutation step and the original formula in Z3 and CVC4. x-axis: mutation step.

fiers of Z3's trace. Here, we under-approximate the similarity of Z3 traces by considering the identifier renaming as the change of the trace.

In our experiment, we fix the trace of the original formula to be t_1, and t_2 corresponds to the trace triggered by the mutant. Figure 4.10 shows the similarity of the corresponding mutation step averaged over all formulas in $\texttt{Seeds}_{1000}$. The results of Z3 and CVC4 consistently show that along with a gradual mutation step increase, the similarity between the traces triggered by the mutant and the original formula gradually decreases. The result indicates that OpFuzz can generate diverse test cases that trigger different execution traces via type-aware operator mutation.

TAKEAWAYS We designed three quantitative evaluations to measure and gain an intuition about the effectiveness of OpFuzz. First, we observe that OpFuzz can find a significant number of bugs in various logics, solver configurations, most of which are in default mode. Second, to understand why OpFuzz can find so many bugs, we designed a coverage evaluation. The evaluation result shows that OpFuzz can increase coverage, but the increment is minor. As the coverage evaluation did not answer why OpFuzz is effective, we further designed the third evaluation investigating the similarity of execution traces. The trace evaluation shows that OpFuzz can gradually change the execution traces of the solvers, which partially explains the effectiveness of OpFuzz.

4.5 in-depth bug analysis

Having extensively evaluated OpFuzz, this section presents an in-depth study on OpFuzz's bug findings. We (1) quantify the fixing efforts for Z3's and CVC4's developers, (2) identify weak components in Z3 and CVC4, and (3) examine the file sizes of bug-triggering SMT formulas. We summarize the insights gained and then present selected bug samples, and examine their root causes, along with the developer's fixes.

4.5.1 *Quantitative Analysis*

We collected all GitHub bug reports that we filed in our extensive evaluation of OpFuzz. This data serves as the basis for our analysis. We guide our analysis with three research questions.

RQ1: How much effort did the developers take to fix bugs found by OpFuzz?

To approach this question, we consider two metrics: the files affected by a bug fix and the number of lines of code (LoC) for fixing. If a bug causes many lines of code and/or file changes, this may indicate a high effort for the developers to fix the bug. To examine the LoC and file changes for the bugs found by OpFuzz, we collected 377 bug fixing commits in Z3 and 101 bug fixing commits in CVC4. We solely considered commits that could be one-to-one matched to their bug reports *i.e.*, the commit log exclusively mentions the issue of our bug report. Figure 4.11 shows the distributions of file changes for bug-fixing commits in Z3 (left) and CVC4 (right). We observe that, in both solvers, most bug-fixing commits change less than five files, and the single file fixes are the majority. However, a few commits affected many files. We have manually examined the right tail of the distribution. We specifically present the top-2 file changing commits in both Z3 and CVC4 individually to demonstrate exemplary reasons for major changes in Z3 and CVC4. We begin with Z3. The highest-ranked bug-fixing commit in Z3 triggered 65 changes. The main part of this fix was in

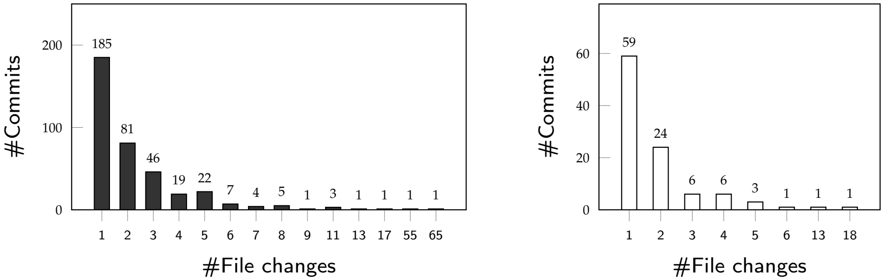


Figure 4.11: Distribution of the number of files affected by a single bug-fixing commit. Z3 (left) and CVC4 (right).

"smt/theory_bv.cpp" which is the implementation of bit-vector logic and also serves as the low-level implementation for floating-point logic. The developers' fix resulted in many function name updates, added checkpoints, and additional 64 file changes. Another bug-fixing commit in Z3 that affected 55 files is a crash. It was caused by an issue in Z3's abstract syntax tree. The core issue addressed by this fix was in "ast/rewriter/rewriter_def.h" and "ast/rewriter/th_rewriter.cpp". Reorganizations of the assertion checks triggered additional 54 file changes. In CVC4, the top-2 fixes with the most file changes (18 and 13) are both caused by crash bugs affecting string operators. The first bug is due to the unsound variable elimination that triggered the assertion violation. The fix refactored the variable elimination with 295 LoC changed. For fixing the second bug, the developers added support for the regex operators `re.loop` and `re.^` that were recently added to the theory of strings. As an intermediate conclusion, we observe that although a relatively high number of file changes indicate extensive revisions in the SMT solvers Z3 and CVC4, their root causes are often rather simple fixes such as updating function names, adding assertions, *etc.* We therefore also investigate the LoC changes for each bug-fixing commit. Many simple fixes, on the other hand, exhibit subtle missed corner cases. However, the higher numbers of changed files are sometimes caused by the updating of the function names, adding new assertions, and so on. Thus, to be comprehensive, we also analyze the commits by their LoC changes. The crash bug leads to the updating of the CVC4 support for new string standard of operators `re.loop` and `re.^`, which changes 13 files. The fix with 6 file changes in CVC4 is also for a crash bug. The one with 55 file changes in Z3 is the fix for a crash bug due to the abstract syntax tree issue. that are related to the formula rewriters. The fix with 13 file changes in CVC4 is

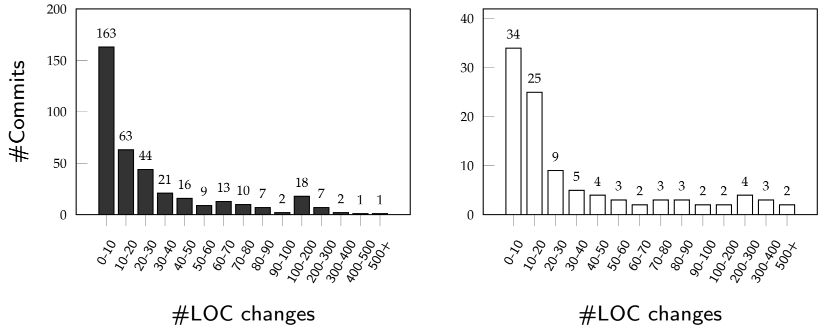


FIGURE 4.12: LoC changed by a single commit in Z3 (left) and CVC4 (right).

related to a crash bug report. It changed various files related to quantifiers. As the investigations show, the relatively higher numbers of changed files indicate the extensive revision of the solvers. However, the higher numbers of changed files are sometimes caused by the updating of the function names, adding new assertions, and so on. Thus, to be comprehensive, we also analyze the commits by their LoC changes.

Figure 4.12 presents the distributions of the LoC changes for each bug-fixing commit. For both Z3 and CVC4, we observe that most commits have less than 100 LoC changes and many bug fixes only involve a 0-10 LoC change. We have manually inspected all 0-10 LoC fixes and observed the majority of them are subtle corner cases. Again we examine top-2 commits for each solver. In Z3 these have 572 and 332 LoC changes respectively. The 572 LoC change commit is a fix for a soundness bug in string logic. It leads to an extensive change in the rewriter of the sequential solver. The 481 LoC changes commit is a fix for a soundness bug in non-linear arithmetic logic. The fix was systematically revamping the decoupling of monomials in non-linear arithmetic logic. For CVC4, the top-2 commits have 1,162 and 588 LoC changes respectively. The 1,162 LoC change in CVC4 commit fixes a crash bug by systematically removing the instantiation propagator infrastructure of CVC4. The developer commented that they will redesign this infrastructure in the future. The bugfix with 588 LoC changes is fixing a soundness bug which is labeled as "major" in CVC4's issue tracker. The bug is due to a buggy ad-hoc rewriter that was incorporated into CVC4's extended quantifier rewriting module. The fix deleted the previous buggy rewriting steps and re-implemented an alternative rewriter. Compared to the analysis of file changes, commits with high LoC have a stronger correlation with interesting and systematic fixes in the SMT solvers. On average, the bugs found by OpFuzz led to 34 and 63 LOC changes for each commit in Z3 and CVC4 respectively.

File	#Commits
smt/theory_seq.cpp	33
smt/smt_context.cpp	30
smt/theory_lra.cpp	25
qe/qsat.cpp	16
ast/ast.cpp	16
smt/theory_arith_nl.h	15
ast/rewriter/seq_rewriter.cpp	14
ast/rewriter/rewriter_def.h	12
tactic/arith/purify_arith_tactic.cpp	11
smt/theory_seq.h	11

(a)

Filename	#LoC changes
smt/theory_seq.cpp	1082
ast/rewriter/seq_rewriter.cpp	837
smt/theory_arith_nl.h	637
smt/theory_lra.cpp	434
tactic/ufbv/ufbv_rewriterċpp	375
math/lp/emonics.cpp	333
smt/smt_context.cpp	265
smt/theory_recfun.cpp	247
tactic/core/dom_simplify_tactic.cpp	229
tactic/arith/purify_arith_tactic.cpp	224

(b)

File	#Commits
theory/arith/nonlinear_extension.cpp	7
theory/strings/theory_strings.cpp	6
preprocessing/passes/unconstrained_simplifier.cpp	5
theory/arith/nl_model.cpp	5
smt/smt_engine.cpp	5
theory/quantifiers/extended_rewriter.cpp	4
theory/quantifiers/quantifiers_rewriter.cpp	4
theory/quantifiers_engine.cpp	3
theory/quantifiers/instantiate.cpp	3
theory/arith/nonlinear_extension.h	3

(c)

Filename	#LoC changes
theory/quantifiers/inst_propagator.cpp	864
theory/quantifiers/quantifiers_rewriter.cpp	611
theory/arith/nonlinear_extension.cpp	519
theory/strings/regexp_operation.cpp	292
theory/quantifiers/local_theory_ext.cpp	270
theory/strings/theory_strings.cpp	250
theory/arith/nonlinear_extension.h	212
preprocessing/passes/int_to_bv.cpp	201
theory/quantifiers/inst_propagator.h	194
smt/smt_engine.cpp	130

(d)

Figure 4.13: Top-10 (a) files affected by bug fixing commits in Z3. (b) LoC changes per file in Z3 (c) files affected by bug fixing commits in CVC4. (d) LoC changes per file in CVC4.

RQ2: Which parts/files of Z3's and CVC4's codebases are most affected by the fixes?

In this research question, we investigate the influence of OpFuzz's bug findings on the respective codebases of Z3 and CVC4. For this purpose, we use two metrics. First, the number of bug-fixing commits that changed a specific file f in either Z3's or CVC4's codebase, *i.e.*, in how many bug-fixing commits file g was included. The second metric is the cumulative number of LoC changes for a file f caused by fixes in either Z3's or CVC4's codebase. For each file f we add up additions and deletions.

In general, there are 103 files in CVC4 and 348 files in Z3 are affected by the fixes of our bugs. Figure 4.13 shows a top-10 ranking of files in Z3's (top row) and CVC4's codebase (bottom row) concerning the two metrics. We observe that in both Figure 4.13a and Figure 4.13b, most files belong to the `"smt"` directory which contains the core implementations of Z3. Strikingly, the files `"smt/theory_seq.cpp"` (Z3's sequence and string solvers), `"smt/theory_arith_nl.h"` (Z3's non-linear arithmetic solver) and `"smt/theory_lra.cpp"` (Z3's linear arithmetic solver) are ranked in the top-6 in both #Commits and #LoC changes rankings. This suggests that many of OpFuzz bug findings lead to the fixes in the core components of Z3. Besides files from the `"smt"` directory, the remaining files are mostly part of Z3's `"tactic"` and `"ast"` directories. These contain the implementations of solver the front-end and Z3's solving tactics. Note, several formulas rewriters related files such as files `"ast/rewriter/seq_rewriter.cpp"`, `"ast/rewriter/rewriter_def.h"` and `"tactic/ufbv/ufbv_rewriter.cpp"` are also highly ranked in the top-10 files affected by the fixes. We now turn our attention to CVC4. Consider the bottom row of Figure 4.13b that presents file and LoC rankings in CVC4. The files that are related to quantifiers (under the path `"theory/quantifiers"`) are the majority in both rankings. Besides, the files `"nonlinear_extension.cpp"`, `"theory_strings.cpp"` and `"quantifiers_rewriter.cpp"` are listed in both rankings. The file `"nonlinear_extension.cpp"` was the implementation of the non-linear arithmetic solver, and a recent pull request moved the core of the non-linear arithmetic solver elsewhere. The file `"quantifiers_rewriter.cpp"` contains the implementations of quantifier rewriters that caused soundness bugs, as RQ1 revealed. The file `"theory_strings.cpp"` contains the decision procedures for string logic in CVC4. Moreover the model generator of non-linear arithmetic (`"nl_model.cpp"`) and the pre-processor (`"int_to_bv.cpp"`, `"unconstrained_simplifier.cpp"`) are also heavily influenced by bug fixes.

RQ3: What is the file-size distribution of the bug-triggering formulas?

In this research question, we investigate the file-size distribution of reduced bug-triggering formulas. We collected the bug-triggering formulas from all confirmed and fixed Z3 and CVC4 bug reports we filed. Figure 4.14 presents the distribution of bug-triggering formulas collectively for Z3 and

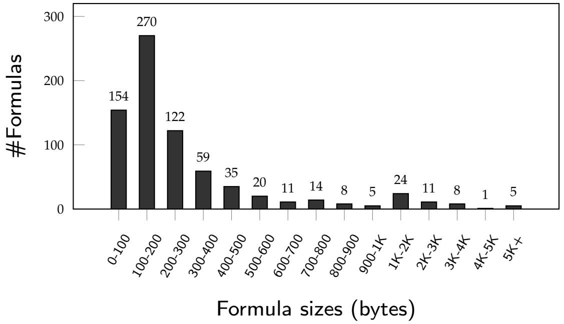


FIGURE 4.14: File-size distribution of reduced bug-triggering formulas.

CVC4. According to Figure 4.14, most formulas have less than 600 bytes, while the range of 100-200 bytes has the highest formula count. The formula with 19,473 bytes triggered a crash bug in CVC4.

The top-3 smallest bug-triggering formulas have 21, 30, and 34 bytes respectively. The 21-byte formula is an invalid formula that triggers a crash bug in Z3. Crash-triggering formulas are 30 bytes and 34 bytes in size for Z3 and CVC4, respectively. These three bugs were all fixed promptly, i.e., in less than one day, which is significantly faster than the bugs triggered by the top-3 largest formulas. The average size of the bug-triggering formulas reported is 426 bytes, which is usually small enough for the developers.

4.5.2 *Insights*

INSIGHT 1: OPFUZZ'S BUGS ARE OF HIGH QUALITY RQ1 and RQ2 have shown that OpFuzz's bug findings have not only led to non-trivial file and LoC changes in both CVC4 and Z3 but also motivated the developers to reorganize and redesign some parts of the solvers. Systematic infrastructure changes such as the decoupling of the monomial instantiation propagator show this. Furthermore, OpFuzz's bugs affected core implementations of the SMT solvers Z3 and CVC4. As RQ2 presented, the `"smt"` directory in Z3 and `"theory"` directory in CVC4, solvers are among the most affected. Besides, the bugs also affected various pre-processors and rewriter components. Third, the bug-triggering formulas that OpFuzz found could be reduced to reasonable file sizes (see RQ3).

INSIGHT 2: WEAK COMPONENTS IN Z3 AND CVC4 From the rankings in RQ2, we identify several "weak" components in Z3 and CVC4. First, in both Z3 and CVC4, source files for the non-linear arithmetic solvers rank high. This indicates: decision procedures for non-linear arithmetic are among the weak components in SMT solvers. Apart from these, rewriters are weak components as well. Z3's "rewriter_def.h", "ufbv_rewriter.cpp" and "seq_rewriter.cpp" are among the top-10 in LoC changes. In CVC4, the quantifier rewriter "quantifiers_rewriter.cpp" is ranked high (5th and 2nd in Figure 4.13c and Figure 4.13d respectively). In Z3, we identified the tactics to be a weak component. Among the filed bug reports, there are 126 including reports related to tactics. In Figure 4.13, these are "purify_arith_tactic.cpp" and "dom_simplify_tactic.cpp" which are ranked 9th or 10th in both Figure 4.13a and Figure 4.13b. In Figure 4.13b, for Z3, the rewriter related files "rewriter_def.h", "ufbv_rewriter.cpp" and "seq_rewriter.cpp" exist in the top-10 rankings. In conclusion, the weak components of the solvers we mentioned above should be rigorously tested (by OpFuzz and further techniques), to uncover the potential issues of the solvers.

INSIGHT 3: BUGS FOUND BY OPFUZZ CAN USUALLY BE REDUCED TO SMALL-SIZED FORMULAS BUT BUG REDUCTION CAN BE CHALLENGING As we have observed (c.f. Figure 4.14), 90% of all bugs found by OpFuzz are triggered by formulas of less than 600 bytes. Small-sized formulas facilitate the bug-fixing efforts significantly. As we observed in RQ3, the top-3 largest formulas took the developers around half a month while the top-3 smallest formulas were fixed very fast, usually within just a few hours. However, reducing SMT formulas to such small sizes can be challenging. ddSMT [81] is the only specialized SMT formula reducer for that purpose which does however not fully support the SMT-LIB 2.6 standard and formulas in string logic. We, therefore, preferred C-Reduce, a C code reducer to reduce SMT formulas. While C-Reduce was effective, bug reduction is challenging, especially for formulas with high solving time.

4.5.3 *Selected Bug Samples*

This subsection details multiple bug samples from our extensive bug-hunting campaign of Z3 and CVC4 and inspects the root causes.

FIGURE 4.15A shows a soundness bug in Z3's bit-vector logic. The formula is clearly unsatisfiable as the nested bvxnor expression equals the

```
(declare-const a (_ BitVec 8))
(declare-const b (_ BitVec 8))
(declare-const c (_ BitVec 8))
(assert (=  (bvxnor a b c)
    (bvxnor (bvxnor a b) c)))
(check-sat)
```

(a) Soundness bug in Z3 caused by a logic in the handling of the ternary `bvxnor`.
https://github.com/Z3Prover/z3/issues/2832

```
(set-logic ALL)
(declare-fun x () Real)
(assert (< x 0))
(assert (not (=
    (/ (sqrt x) (sqrt x)) x)))
(check-sat)
```

(b) Soundness bug in CVC4: inadmissible reduction of the square root operator.
https://github.com/CVC4/CVC4/issues/3475

```
(declare-fun a () Int)
(declare-fun b (Int) Bool)
(assert (b 0)) (push)
(assert (distinct true
    (= a 0) (not (b 0))))
(check-sat)
```

(c) Soundness bug in Z3 in the boolean rewriter handling the distinct operator.
https://github.com/Z3Prover/z3/issues/2830

```
(set-logic QF_AUFBVLIA)
(declare-fun a () Int)
(declare-fun b (Int) Int)
(assert (distinct (b a)
    (b (b a))))
(check-sat)
```

(d) Soundness bug in CVC4 due to a variable re-use in a simplification.
https://github.com/CVC4/CVC4/issues/4469

```
(declare-fun a () String)
(declare-fun b () Int)
(assert (> b 0))
(assert (= (int.to.str b)
    (str.++ "0" a)))
(check-sat)
```

(e) Soundness bug in Z3: a missing axiom in the integer to string conversion.
https://github.com/Z3Prover/z3/issues/2721

```
(declare-fun x () String)
(declare-fun y () String)
(assert (= (str.indexof x y 1)
    (str.len x)))
(assert (str.contains x y))
(check-sat)
```

(f) Soundness bug in CVC4 due to an invalid indexof range lemma.
https://github.com/CVC4/CVC4/issues/3497

```
(declare-fun a () Real)
(assert (forall ((b Real))
    (= (= a b) (= b 0))))
(check-sat-using qe)
```

(g) Longstanding soundness bug in Z3's qe tactic (since version `4.8.5`).
https://github.com/Z3Prover/z3/issues/4175

```
(declare-fun a () Real)
(assert (= (* 4 a a) 9))
(check-sat)
(get-model)
```

(h) Invalid model bug in CVC4: incorrect implementation of the square root.
https://github.com/CVC4/CVC4/issues/3719

```
(declare-fun a () Int)
(declare-fun b () Real)
(declare-fun c () Real)
(assert (> a 0))
(assert (= (* (/ b b) c) 2.0))
(check-sat)
(check-sat)
(get-model)
```

(i) Invalid model bug in Z3.
https://github.com/Z3Prover/z3/issues/3118

```
(declare-fun d () Int)
(declare-fun b () (Set Int))
(declare-fun c () (Set Int))
(declare-fun e () (Set Int))
(assert (subset b e))
(assert (= (card b) d))
(assert (= (card c) 0 (mod 0 d)))
(assert (> (card (setminus e
    (intersection (intersection e b)
    (setminus e c)))) 0))
(check-sat)
```

(j) Soundness bug in CVC4's set logic.
https://github.com/CVC4/CVC4/issues/4391

Figure 4.15: Selected bug samples in Z3 and CVC4.

unnested `bvxnor` expression. However, Z3 reports `unsat` on it, which is incorrect. The root cause for this bug is an incorrect handling of the ternary `bvxnor` in Z3's bitvector rewriter "`bv_rewriter.cpp`". The `bvxnor` was implemented as the negation of the `bvxor` operator. This is correct in the binary case, however, incorrect for the n-ary case. To see this consider, *e.g.*, for the assignment `a = b = c = true`:

$$\texttt{(bvxnor a b c)} \neq \texttt{(not (bvxor a b c))}$$
$$\texttt{(bvxnor (bvxnor true true) true)} \neq \texttt{(not (bvxor (bvxor true true) true))}$$
$$\texttt{true} \neq \texttt{false}$$

In the fix, Z3 developers recursively reduces n-ary `bvxnor` expression to the binary case. The fix led to a 17 LoC change in `ast/rewriter/bv_rewriter.cpp`.

FIGURE 4.15B shows a soundness bug in the implementation of the symbolic square root in CVC4. The formula can be satisfied by assigning an arbitrary negative real to variable `x`. CVC4 incorrectly reported `unsat` on this formula. The root cause for this bug is an inadmissible reduction of the square root expression `(sqrt x)` to "choice real `y` s.t. $\texttt{x} = \texttt{y} \cdot \texttt{y}$". For negative `x`, there is no `y` to satisfy the equation. However, square roots of negative numbers are permitted by the SMT-LIB standard. CVC4's developers fixed this bug by interpreting square roots of negative numbers as an undefined value that can be chosen arbitrarily. For the formula in Figure 4.15b, the term `(/ (sqrt x) (sqrt x))` can be arbitrarily chosen, as the second assert demands `x` to be negative. Therefore, the formula in Figure 4.15b is satisfiable. The bug-fixing pull request was labeled as "major" which reveals that this issue was of high importance to the CVC4 developers. The fix led to a 126 LoC change on 5 files.

FIGURE 4.15C is a soundness bugs in Z3. Although the second assert is unsatisfiable (as `true` cannot be distinct with `(not (b 0))`, Z3 reported `sat` on this formula. The bug is caused by a logic error in a loop condition of a rewriting rule for the `distinct` operator. An incorrect index condition accidentally skips the last argument in an n-ary distinct. The `push` command is necessary for triggering the bug, as it actives the rewriter for `distinct`. The developer has fixed this bug by correcting the index condition. Hence, his fix consisted of only two character deletes in `ast/rewriter/bool_rewriter.cpp`.

FIGURE 4.15D shows a soundness bug in CVC4's logic of uninterpreted functions In default mode, CVC4 incorrectly reports `unsat` on this satisfiable formula. If we disable unconstrained simplification (`-no-unconstrained-simp`), CVC4 correctly reports `unsat`. The bug is caused by an unsound variable reuse. Our bug report got a "major" label from CVC4's developers and was promptly fixed. The fix consists of 3 LoC deletions in the unconstrained simplifier implementation "`preprocessing/passes/unconstrained_simplifier.cpp`".

FIGURE 4.15E depicts a soundness bug in Z3's QF_SLIA logic. The formula is unsatisfiable, since if assertion `b > 0` holds, there does not exist an a starting with `"0"`. However, Z3 reports `sat` on this formula. The developers fixed this issue by adding an axiom.

FIGURE 4.15F shows a soundness in CVC4's string logic. The intuition behind this formula is the following. The index of string `y` in `x` after position 1 should be equal to the length of string `x`. Furthermore `x` should contain `y`. The formula can be satisfied by setting `y` to the empty string and `x` to a string of length 1. However, CVC4 incorrectly reports `unsat`. The root cause was a logic error in `theory/strings/theory_strings.cpp`. The developer's fix changed three characters in `theory/strings/theory_strings.cpp`.

FIGURE 4.15G presents a long-standing soundness bug in Z3's `qe` tactic. It affects z3 release from version `4.8.5` to `4.8.7`. The `qe` tactic is an equisatisfiable transform for eliminating quantifiers. Hence, the satisfiability should not be changed by using the `qe` tactic. The formula is satisfiable by assigning `a` to `0`, while Z3's `qe` tactic reports `unsat`. The bug has been confirmed by Z3's developers but has not been fixed yet.

FIGURE 4.15H shows an invalid model bug in CVC4. CVC4 correctly reports `sat` but generates the model $\{a \mapsto \frac{-9}{2}\}$ which does not satisfy the formula. The bug is caused by CVC4's implementation of the square root. A logic error assigns the result of the square root to be the square root's argument. The fix is labeled as "major" by the developers, and promptly fixed only with a two LoC change in file `theory/arith/nl_model.cpp`.

FIGURE 4.15I shows an invalid model bug in Z3. The `(check-sat)` command appears twice in the formula. This means that Z3 is queried twice for solving. Z3 reports `unknown` for the first query and `sat` for the second. In the second query, Z3 gives the following invalid model $\{a \mapsto 0, b \mapsto 0.0, c \mapsto 16.0, \frac{0}{0} \mapsto \frac{1}{8}\}$ violating `(> a 0)`. The developers fixed this bug through three LoC changes in file `solver/tactic2solver.cpp`.

FIGURE 4.15J presents a soundness bug in CVC4's set logic. CVC4 returns `unsat` on this satisfiable formula. The root cause is an incorrectly implemented set cardinality rule in the cardinality extension of CVC4. CVC4's set solver uses lemmas to guess the equalities for terms by identifying cycles of terms $e_1 = \cdots = e_2 = \cdots = e_2$. CVC4 has incorrectly assumed that these cycles are loops and in that case, would conclude $e_1 = \cdots = e_2$. However, the cycles could have a lasso form which was triggered by our formula. The developers fixed this issue, included the formula to CVC4's regression test suite, and marked the pull request to be critical for CVC4's 1.8 release. The fix was labeled as "major" and made 9 LoC changes `theory/sets/cardinality_extension.cpp`.

4.6 RELATED WORK

TESTING SMT SOLVERS This is not the first work on testing SMT solvers. Roughly ten years ago, the fuzzing tool FuzzSMT [60] was proposed, which is based on differential testing and targeted bit-vector logic. FuzzSMT uses a grammar for generating the SMT formula. FuzzSMT totally found 16 solver defects in five solvers, however, none in Z3. BtorMBT [61] is a testing tool for Boolector [62], an SMT solver for the bit-vector theory. BtorMBT tests Boolector by generating random valid API call sequences.

The efforts of the SMT-LIB initiative [47] have resulted in formalized SMT theories and common input/output file formats. In addition, the yearly solver competition SMT-COMP heavily penalized solvers with soundness issues. Consequently, SMT solvers have robustified and finding bugs in SMT solvers became more difficult. Researchers have hence targeted the less mature logics such as the recently proposed theory of strings. Blotsky *et al.* [56] proposed StringFuzz which focuses on performance issues in string logic. StringFuzz generates test cases in two ways, one is mutating and transforming the benchmarks, and another one is randomly generating formulas from a grammar. StringFuzz found 2 performance bugs and 1 implementation bugs in z3str3. Bugariu and Müller [63] proposed a formula synthesizer for String formulas that are by construction satisfiable or unsatisfiable. They showed that their approach can detect many existing bugs in String solvers and they found 5 new soundness/incorrect model bugs in z3 and z3str3. However, it remained an open question whether automated testing tools could find bugs in theories except for the unicode string theory in Z3 and CVC4. Recently, semantic fusion [1] has been proposed which is an approach to stress-test SMT solvers by fusing formula pairs that are by construction either satisfiable or unsatisfiable. Winterer, Zhang, and Su's tool YinYang found a few dozen bugs in Z3 and CVC4. STORM [32], another mutation-based SMT solver testing approach, found 27 bugs in Z3, however none in CVC4. Another related approach is BanditFuzz [58], an RL-based fuzzer to detect SMT solver performance issues.

Compared to previous work, type-aware operator mutation is the simplest, while it has also been demonstrated to be the most effective technique for testing SMT solvers. Type-aware operator mutations show a promising direction for testing SMT solvers which can benefit the whole community.

TESTING PROGRAM ANALYZERS SMT solvers are fundamental tools for various program analyzers. Hence, bugs in SMT solvers may affect the reliability of program analyzers. Especially because program analyzers have become mature for practical use in recent years, ensuring the reliability of program analyzers is crucial [48]. There are several works on program analyzer's robustness. For example, Zhang *et al.* [64] tested software model checkers via reachability queries, Bugariu *et al.* [68] found soundness and precision bugs in numerical abstract domains, Qiu *et al.* [69] and Pauck *et al.* [70] found bugs in the analyzers of Android apps. Type-aware operator mutation contributes to testing program analyzers by finding bugs in SMT solvers. Differential testing-based approaches have been effective in finding bugs in program analyzers. For example, Klinger *et al.* [65] and Kapus and Cadar [66] proposed the approaches for testing software model checkers and symbolic executors respectively using differential testing, Wu *et al.* [67] tested alias analyzers by cross-checking the dynamic aliasing information. OpFuzz likewise uses differential testing to detect soundness bugs.

MUTATION-BASED TESTING Type-aware operator mutation belongs to the family of mutation-based testing techniques. The closest work is skeletal program enumeration (SPE) [82], an approach for validating compilers. Similar to type-aware mutation testing, program skeletons are generated from a set of seed programs. The holes in these skeletons are then systematically filled by exhaustive enumeration. However, unlike type-aware operator mutation, SPE focuses on program variables and not on functions. SPE provides relative guarantees with respect to the input seed programs.

Type-aware operator mutation is also related to FuzzChick [83], a coverage-guided fuzzer for Coq programs. FuzzChick generates test cases by semantic mutations at the type level. FuzzChick is aware of parameter types and generates new values for the parameters while preserving type-correctness.

Type-aware operator mutation, on the other hand focuses on the operators' types and to generate highly diverse SMT formulas.

Type-aware operator mutation also belongs to black-box fuzzing techniques. The black-box fuzzing techniques, such as SYMFUZZ [84], leverage user-provided seeds and generate new mutated inputs to uncover security issues. Grey-box fuzzing enhances black-box fuzzing by code coverage guidance and has been successfully applied to software testing recently. AFL [85] is a popular tool for binary grey-box fuzzing. Follow-up works, such as FairFuzz [86] and Steelix [87], improved the performance of AFL on the binary level. However, binary-level fuzzing is ineffective on programs with highly structured inputs (*e.g.* PDF viewers, programming language engines *etc.*) because of the many syntactically invalid inputs being generated. To generate valid test inputs, grammar-aware grey-box fuzzers were proposed. AFLSmart [88], Superion [89] and Nautilus [90] are general grammar-aware grey-box fuzzers targeting programming language engines. They use code coverage to guide the grammar-aware mutations. As a key difference to type-aware operator mutation, they both need to fully parse the program and work on the abstract syntax tree level, which may lead to a higher computational cost during fuzzing. Type-aware operator mutation, on the other hand, works on the token level and without fully parsing the formula.

Besides general black-box and grey-box fuzzing, various domain-specific fuzzing approaches exist, e.g., for testing compilers [72, 75, 82, 91, 92], testing database management systems [93, 94, 95, 96], and testing OS kernel [97, 98, 99]. Type-aware operator mutation is also a domain-specific fuzzing technique that is unusually effective for testing SMT solvers.

5

GENERATIVE TYPE-AWARE MUTATION

This chapter presents *Generative Type-Aware Mutation*, a hybrid of mutation-based and grammar-based fuzzing realized by our tool TypeFuzz. TypeFuzz generalizes OpFuzz (Chapter 4) and can grow and shrink SMT formulas overcoming a major limitation of OpFuzz. TypeFuzz is a highly effective fuzzer: it found several long-latent soundness bugs in CVC4 that has proven to be a very stable solver and has resisted several fuzzing campaigns.

5.1 MOTIVATION

Researchers devised several SMT solver fuzzers and a few large-scale fuzzing campaigns on SMT solvers are ongoing. One such approach is OpFuzz (Chapter 4) which found several hundreds of bugs in the SMT solvers Z3 and CVC4. However, despite its effectiveness, OpFuzz has several limitations. First, OpFuzz is limited by its finite mutation space: the seed formulas have a fixed set of operators and for each of them, there are often only 2-3 choices for mutation. Furthermore, as fuzzing campaigns and especially OpFuzz have led to hundreds of bug fixes in the SMT solvers Z3 and CVC4, the solvers have matured. Because of this effect [100], fuzzers are finding progressively fewer critical bugs. Yet, important bugs are missed.

Consider the formula in Figure 5.1 which manifests a long-latent soundness bug in CVC4. The "declare-fun" statements specify two string variables and one integer variable respectively, the "assert" specifies the constraints, and the "check-sat" queries the solver. The formula is satisfiable which Z3 correctly reports. However, CVC4 returns `unsat` on this formula which indicates a soundness bug in CVC4. This bug is long-latent: it existed in CVC4 since at least cvc4-1.5—for almost four years. [1] Moreover, since March 2019, several large-scale fuzzing campaigns have targeted string logic (some even exclusively), yet none of the other fuzzers found this bug. It is a refutational soundness bug—the most critical bug category.

We introduce *Generative Type-Aware Mutation*, a novel, effective approach for testing SMT solvers, capable of finding many longstanding soundness bugs in both Z3 and CVC4. It has found the almost four-year latent sound-

[1] cvc4-1.5 was released on July 10, 2017. This bug does trigger in all releases but cvc4-1.6.

```
(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () Int)
(assert (= "B" (str.replace (str.substr "A" 0 z) ""
        (str.replace "B" (str.substr "B" 0 0) (str.substr "A" 0 z)))))
(check-sat)
```

Figure 5.1: Almost four-year latent soundness bug in CVC4's string logic. https://github.com/cvc5/cvc5/issues/5940

ness bug of Figure 5.1. Moreover, with Generative Type-Aware Mutation, we reported 322 bugs in the state-of-the-art SMT solvers Z3 and CVC4, 290 bugs were confirmed and 278 bugs were fixed. Most notably, Generative Type-Aware Mutation found 20 soundness bugs in CVC4's default mode alone. Several of them (7/20) were at least 2 years latent and predated all previous SMT solver fuzzing campaigns.

By comparison, prior approaches did not find any bugs in CVC4 [32, 63] and others found similar numbers of soundness bugs during much longer time spans: YinYang [101] found eight in nine months, and OpFuzz [2] found eleven in a year. All approaches were reportedly using the SMT-LIB seeds and similar resources as TypeFuzz did. TypeFuzz found these bugs despite robustified Z3 and CVC4 thanks to the bug fixes that resulted from prior fuzzing campaigns. The core idea behind *Generative Type-Aware Mutation* is simple: to combine mutational and grammar-based type-aware fuzzing. Given a seed formula φ, we first choose an expression *expr* within φ. Second, we pick an operator *op* of the same type as *expr* and fill *op*'s arguments with expressions from φ. The newly generated expression then replaces *expr* in φ. The formula φ_{mutant} is then used to test SMT solvers.

5.2 ILLUSTRATIVE EXAMPLE

The key idea of Generative Type-Aware Mutation is mutating expressions in the AST of an SMT-LIB script by newly generated expressions of the same type. Let φ be a seed formula (see Figure 5.2).

STEP 1 CHOOSE A RANDOM EXPRESSION: We first choose a random expression $expr_1$ from the set of φ's expressions $expr(\varphi)$. Say we have picked the $expr_1 =$ `x`. The expression is of type `String` and will serve as the replacee for the newly generated expression.

```
(declare-fun x () String)
(assert (> (- (str.to_int
    (str.++ x x))) 0))
(check-sat)
```

$expr_1$ ∈ { `x` , `0`, `(str.++ x x)`,$\cdots$}

(a) Choose random expression

op ∈ { `str.from_int` , `str.++`, $\cdots$}

```
(str.from_int Int String)
```

`int` ∈ { `0` , `(str.to_int (str.++ x x))`,
`(- (str.to_int (str.++ x x)))`}

(b) Choose operator & integer expression

$$expr_2 = (\overset{op}{\texttt{str.from\_int}}\ \overset{\texttt{int}}{\texttt{0}})$$

(c) Generate new expression

```
(declare-fun x () String)
(assert (> (- (str.to_int
    (str.++ (str.from_int 0) x))) 0))
(check-sat)
```

(d) Mutant formula φ_{mutant} (Z3#5108)

FIGURE 5.2: Generative Type-Aware Mutation illustrated.

STEP 2 CHOOSE A RANDOM OPERATOR: Next, we choose a suitable random operator. Such an operator should have the return type `String` and for all of its arguments, there should be at least one expression of conforming type in $expr(\varphi)$. Since φ contains terms of type `Bool`, `Int`, and `String`, the operator's arguments should be one of those types. Candidates are the string to integer conversion function `str.from_int`, the string concatenation `str.++`, and all other operators taking `Bool`, `String` as arguments and returning `Bool`. For the complete list of possible candidates that we use, we refer the reader to Section 5.4. Assume we have chosen the operator `str.from_int`.

STEP 3 GENERATE NEW EXPRESSION: Then, we generate an expression $expr_2$ with respect to the signature of the chosen operator. The signature for the operator `str.from_int` is defined as

```
(str.from_int Int String)
```

Hence, we select an `Int` expression from $expr(\varphi)$. For the single parameter of type `Int`, we choose `0`. Then, with the chosen operator and expression, we construct the following new expression:

$$expr_2 = \texttt{(str.from\_int 0)}$$

STEP 4 SUBSTITUTION: Finally, we substitute $expr_1$ by $expr_2$ in φ which results in the formula φ_{mutant}. We feed φ_{mutant} to two or more SMT solvers and compare their results.

The formula φ_{mutant} is a real case. Z3 and CVC4 give different results on φ_{mutant}. CVC4 correctly reported `sat` on it, while Z3 incorrectly reported `unsat`. We have filed this bug on the Z3 issue tracker. The developers promptly fixed this soundness issue in the trunk version of Z3. As we will show, Generative Type-Aware Mutation is a powerful generalization of type-aware operator mutation and FuzzChick. Neither approach could have generated this bug-triggering formula from the seed φ.

5.3 GENERATIVE TYPE-AWARE MUTATION

This section (1) formally introduces Generative Type-Aware Mutation, (2) shows the conditions under which Generative Type-Aware Mutation produces type-correct formulas, (3) clarifies the relationships to type-aware operator mutation and FuzzChick, and (4) proposes TypeFuzz, a practical fuzzing tool for stress-testing SMT solvers.

DEFINITIONS We use standard notions of typed higher-order logic, such as term, quantifier, and function and write expressions for term occurrences. We view formulas as abstract syntax trees of typed expressions. Such an expression $expr$ has an associated type $type(expr)$ and the set of all types is $types = \{\texttt{Bool}, \texttt{Int}, \texttt{Real}, \texttt{String}, \texttt{RegLan}, \texttt{A}\}$ where `A` is a generic supertype of all the other types. For an expression $expr$ in φ, we define $locals(expr, \varphi)$ to be the set of local variable occurrences in $expr$. When φ is clear from the context, we simply write $locals(expr)$. Within a formula φ, local variables can be defined by quantifiers, let expressions, *etc.* By $expr(\varphi)$, we denote φ's (enumerated) expression occurrences. We write $\varphi[expr_2/expr_1]$ for the substitution of expression $expr_1$ by expression $expr_2$ in φ. An operator op has the attributes $rtype(op)$ and the tuple $arg\_types(op)$ for op's return type and the types of op's arguments. We denote the set of operators by ops and ops_{type} for all operators of return type $type$. The type skeleton $skeleton(\varphi)$ of φ is a tree where each expression $expr$ in φ is represented by its type.

REGULAR TREE GRAMMAR A *regular tree grammar* $G = (N, \Sigma, S, P)$ consists of a finite set of nonterminals N, a ranked alphabet Σ, a starting nonterminal S in N, and a finite set of productions P. Each symbol in Σ has an associated arity, and the productions in P are of the form $A \to t$

where $A \in N$ and $t \in T_\Sigma$ with T_Σ being the set of all trees composable from symbols $\Sigma \cup N$. The language $L(G)$ generated by G describes any tree that can be derived from S using the rule set P.

Definition 5.3.1 (Generative Type-Aware Mutation). Let $G_{\mathsf{GTA}} = (N, \Sigma, S, P)$ be a regular tree grammar and φ a formula:

- $N = types$
- $\Sigma = expr(\varphi)$
- $S = skeleton(\varphi)$
- $P = P_{expr} \cup P_{gen}$ where
  - $P_{expr} = \{type \rightarrow expr \mid expr \in expr(\varphi) \wedge type(expr) = type\}$
  - $P_{gen} = \{type \rightarrow (op\ arg\_types(op)) \mid\ op \in ops_{type}\}$

Formula $\varphi_{mutant} \in L(G_{\mathsf{GTA}})$ is called a **generative type-aware mutant**.

A generative type-aware mutant φ_{mutant} of φ can be conveniently fabricated by replacing an expression $expr_1$ within φ with an expression $expr_2$ which is either another type-conforming expression from $expr(\varphi)$, or rooted with a new operator of type-conforming return type and type-conforming arguments from $expr(\varphi)$. Generative type-aware mutations will by design not lead to type-incorrect replacements, *e.g.*, replacing an integer expression with a string expression. However, as the following example illustrates, they also do not guarantee well-typed formulas. Consider the following formula in SMT-LIB language:

$$\varphi = \texttt{(and (> x 10) (forall ((z Int)) (< z y)))}$$

We choose $expr_1 = $ `(> x 10)` and $expr_2 = $ `(< z y)` to replace $expr_1$ by $expr_2$ in φ. The resulting formula φ_{mutant} is a generative type-aware mutant of φ:

$$\varphi_{mutant} = \texttt{(and (< z y) (forall ((z Int)) (< z y)))}$$

However, φ_{mutant} is not well-typed since the variable `z` is out of the scope of the quantifier. We address this issue by the following definition.

Definition 5.3.2 (local compatibility). Let $expr_1$ and $expr_2$ be two expressions of the same type. We say $expr_2$ is **locally compatible** with $expr_1$ if $locals(expr_2) \subseteq locals(expr_1)$.

Checking for local compatibility avoids the above issue: $locals(expr_1) = \emptyset$ and $locals(expr_2) = \{\texttt{z}\}$ and hence $expr_2$ would not be locally compatible with $expr_1$ preventing the ill-typed formula.

Proposition 5.3.1. *Generative type-aware mutants are well-typed if for every substitution local compatibility is ensured.*

5.3.1 Relationships to FuzzChick and Operator Mutation

This section clarifies the relationships between Generative Type-Aware Mutation and two related techniques that have been used for stress-testing software, FuzzChick and type-aware operator mutation.

FUZZCHICK'S MUTATOR: FuzzChick [83] tests Coq programs using grammar-based generators and coverage feedback. Adapted to our formal setting, this corresponds to $\varphi_{mutant} = \varphi[expr_2/expr_1]$ where $expr_1$ and $expr_2$ are expressions from $expr(\varphi)$.

TYPE-AWARE OPERATOR MUTATION: OpFuzz [2] realizes type-aware operator mutation, an effective technique for SMT solver testing. The key idea is to mutate operators of conformant type. Let φ be an SMT formula and let op_1 of type $type_1$ be one of φ's operator and op_2 of type $type_2$ be an operator of the SMT-LIB specification. $\varphi_{mutant} = \varphi[op_2/op_1]$ is a type-aware operator mutant if $type_2$ is a subtype of $type_1$.

Let formula φ be a formula and $G_{\mathsf{GTA}} = (N, \Sigma, S, P_{gen} \cup P_{expr})$ the regular tree grammar specifying generative type-aware mutations for φ. FuzzChick's mutator can be described by the grammar $G_{\mathsf{FC}} = (N, \Sigma, S, P_{expr})$. Since the grammar G_{FC} is identical to G_{GTA} without generation rules, $L(G_{\mathsf{FC}})$ is a subset of $L(G_{\mathsf{GTA}})$. Every type-aware operator mutation $\varphi_{mutant} = \varphi[op_2/op_1]$ with op_1, op_2 from ops can be imitated by G_{GTA} by starting from the $skeleton(\varphi)$ and applying the productions P_{expr} to generate φ except for op_1. Then, we apply the production $op_1.type \rightarrow (op_2\ type_1, \cdots, type_m)$ from P_{gen} and again rules from P_{expr} to recover the former arguments of op_1.

Corollary 5.3.1. *Generative Type-Aware Mutation generalizes FuzzChick.*

Corollary 5.3.2. *Generative Type-Aware Mutation generalizes Opfuzz.*

The following example is a real case that constructively shows the strict dominance of Generative Type-Aware Mutation over the type-aware operator mutation, the state-of-the-art approach for fuzzing SMT solvers.

Example 5.3.1. Consider formula φ from Figure 5.3a. Assume, we picked the expression $expr_1 =$ `(str.replace x "B" (str.++ "B" "B"))` from $expr(\varphi)$. The expression is of type `String` and will serve as the replacee for the newly generated expression. Next, we choose the operator `str.replace` that takes three strings and returns a string. Then, we generate an expression:

```
(str.replace String String String String)
```

```
(declare-fun x () String)
(declare-fun y () String)
(assert
(= (str.replace x "B" (str.++ "B" "B"))
(str.++ y "B")))
(check-sat)
```

(a) Seed formula φ

```
(declare-fun x () String)
(declare-fun y () String)
(assert
(= (str.replace
(str.replace x "B" (str.++ "B" "B"))
"B" (str.++ y "B"))
(str.++ y "B")))
(check-sat)
```

(b) Mutant formula φ_{mutant} (CVC4 #5915)

FIGURE 5.3: Generative Type-Aware mutation illustrated.

by substituting the function arguments with type-aware expressions from $expr(\varphi)$ such as the following:

$expr_2 =$ `(str.replace (str.replace x "B" (str.++ "B" "B")) "B" (str.++ y "B"))`

Finally, we substitute $expr_1$ by $expr_2$ in φ which results in the formula φ_{mutant} (Figure 5.3b). We feed φ_{mutant} to two or more SMT solvers and compare their results. Z3 and CVC4 give different results on φ_{mutant}. Z3 correctly returned `unsat` on it while CVC4 returned `sat` on it. We have reported this bug to the CVC4 issue tracker. The developers promptly fixed this four-year longstanding soundness issue in CVC4. Neither, type-aware operator nor type-aware expression mutation can find this bug from the seed φ. Type-aware operator mutation cannot generate this bug since the number of operators has increased from φ to φ_{mutant}.

DIFFERENCES IN PRACTICE As a key difference to FuzzChick, generative type-aware mutation can leverage all available operators from the SMT-LIB specification for the types in its skeleton. In contrast, FuzzChick is limited to the operators occurring in the seed formula. If the seed formula contains all operators from the SMT-LIB specification for all the types in its skeleton, the two techniques are identical, i.e, the grammars $L(G_{\mathsf{GTA}})$ and $L(G_{\mathsf{FC}})$ would induce the same language. However, in practice, it is rarely the case that all operators from the specification occur in a formula.

In contrast to type-aware operator mutation, generative type-aware mutation can generate expressions rooted by operators from the SMT specification with type-conforming terms from the seed file as its arguments while type-aware operator mutation can only mutate operators. Hence, generative

Algorithm 5 TypeFuzz's pseudocode

```
1:  procedure TypeFuzz(solvers, seeds)
2:      all_ops ← ReadConfigfile()
3:      triggers ← []
4:      while no termination criterion is met do
5:          $\varphi$ ← random.choice(seeds)
6:          expressions ← GetTypedExpressions($\varphi$)
7:          for i to n do
8:              $\varphi_{mutant}$, success ← GenerativeTypeAwareMutate($\varphi$, expressions)
9:              if not success then
10:                 continue
11:             if not Validate($\varphi_{mutant}$, solvers) then
12:                 triggers ← triggers.append($\varphi_{mutant}$)
13:             $\varphi$ ← $\varphi_{mutant}$
```

type-aware mutation's additional expressive power translates to practical advantages over both techniques.

5.3.2 *TypeFuzz*

Based on Generative Type-Aware Mutation, we have engineered TypeFuzz, a bug-hunting tool. This subsection details the procedures of TypeFuzz.

MAIN PROCESS Algorithm 5 presents the parameterized pseudocode of TypeFuzz. The main process takes as inputs: a set of SMT solvers, *solvers* under test, and a set of seed formulas *seeds*. First, the algorithm reads a configuration file (Line 2). The configuration file (see Figure 5.4) contains all signatures of the SMT-LIB operators and can be customized by the user. The list *triggers* is used for collecting the bug triggers and is initialized to the empty list (Line 3). The body of the while loop is executed until a termination criterion is met. This could be a timeout or an interruption by the user. We first randomly chose a formula φ from *seeds* (Line 5). Then, we call the function GetTypedExpressions (Line 6) which returns the list *expressions*. The body of the for loop (Line 7) realizes n consecutive generative type-aware mutations. At the end of each iteration (Line 13), we reset φ to the previously mutated formula φ_{mutant} to realize the mutation

Algorithm 6 Generative Type-Aware Mutation's pseudocode

```
function GenerativeTypeAwareMutate($\varphi$, $expressions$)
    $unique\_expr \leftarrow$ GetUniqueExpressions($expressions$)
    for $j$ to $len(expressions)$ do
        $expr_1 \leftarrow random.choice(expressions)$
        $expr_2 \leftarrow$ GetReplacee($expr_1, unique\_expr$)
        if $expr_2 \neq None$ then
            return $\varphi[expr_2/expr_1]$, $true$
    return $None, false$

function GetReplacee($expr$, $unique\_expr$)
    $ops \leftarrow \{op \in all\_ops \mid rtype(op) = expr.type\}$
    $op \leftarrow random.choice(ops)$
    $args \leftarrow []$
    for $type$ in $op.arg\_types$ do
        $choices \leftarrow \{e \in unique\_expr | e \neq expr \wedge e.type = type \wedge local\_comp(e, expr)\}$
        if $choices = \emptyset$ then
            return $None$
        $arg \leftarrow random.choice(choices)$
        $args.append(arg)$
    return $make\_expr(op, args)$
```

chain. For parameter n, we have used values in the range of 10 to 100. Smaller n help traverse the seed set faster, larger n lead to deeper mutations. Inside the for-loop body, we first call the function GenerativeTypeAwareMutate which returns the boolean *success* indicating whether the function successfully generated a mutant formula φ_{mutant}. If the mutation attempt was unsuccessful, we continue with the next iteration. Otherwise, we call the function validate with the mutant formula φ_{mutant} and the set of solvers, *solvers*. validate sequentially executes each solver on φ and checks for (1) crashes, *i.e.*, segmentation faults, assertion violations by matching standard output to a known list of errors, (2) soundness issues by comparing the satisfiability results of the solvers, and (3) invalid models where the solver returns an incorrect model on a satisfiable formula. In any of the three cases, the function returns *false* and we add φ_{mutant} to the candidate bugs.

GENERATIVE TYPE-AWARE MUTATION Algorithm 6 presents the implementation of a generative type-aware mutation. The function GenerativeTypeAwareMutate (Line 1) takes a formula and *expressions* as its inputs. We first retrieve the set of unique expressions from the list *expressions*. The list of expressions may contain duplicates since syntactically equivalent expressions can occur multiple times in φ (Line 2). In the for loop (Line 3), we first choose a random expression $expr_1$ from the list of expression *expressions*. Then, we call the function GetReplacee to obtain an expression for replacing $expr_1$. If the function was unsuccessful, it returns *None*. If successful, we return a formula in which $expr_1$ is replaced by $expr_2$ and *true*(Line 7), indicating that the mutation attempt was successful. Otherwise, if after $len(expressions)$ tries no replacee has been found, we return *None* and *false*(Line 8) indicating that the mutation attempt was unsuccessful. The GetReplacee function (Line 9) realizes a greedy algorithm for finding a suitable replacee expression for a given expression *expr*. First, we collect a set of operators of conforming return type with *expr* (Line 10) and randomly choose one of them (Line 11). We then iterate through the argument types of the chosen operator (Line 13). For each argument type, we compute the set *choices* (Line 14) representing the type matching expressions *e* distinct from *expr* that are locally compatible with *expr*. If *choices* is empty, we return *None* (Line 14) to indicate that the mutation attempt was unsuccessful, i.e., there is no expression *e* of the same type that is locally compatible with *expr* and syntactically different. Otherwise, we randomly choose an argument from *choices* and add it to the list of arguments for the chosen operator (Line 17 +

18). In Line 19, we then instantiate the chosen operator with the selected arguments and return them.

COMPUTING LOCAL COMPATIBILITY To realize local compatibility (see Line 14), we use a recursive procedure. For an expression *expr*, we recursively collect the local variables defined by its parent and upward. The reason for collecting local variables from the parent onwards is when the expression declaring the local variable, which itself contains the local variable is substituted by one of its child expressions with the local variable it declared, the mutant will be faulty as the declaration is lost and the local variable becomes undefined.

IMPLEMENTATION We have implemented TypeFuzz on top of the SMT solver fuzzer yinyang [101]. For that matter, we implemented the Generative Type-Aware Mutation as a mutation strategy (260 LoC) and augment the yinyang framework by a type-checker (790 LoC). TypeFuzz's mutations can be customized with a configuration file. We have used the configuration file from Figure 5.4. Its syntax is similar to the meta-language of SMT-LIB theory specifications. [2] We hope this will facilitate SMT developers and practitioners to run customized configurations and have released our tool on GitHub. [3] The tool can be installed via `pip install yinyang`.

5.4 EMPIRICAL EVALUATION

This section details our extensive evaluation with TypeFuzz demonstrating the practical effectiveness of Generative Type-Aware Mutation for testing SMT solvers. Between end of January 2021 and mid September 2021, we were running TypeFuzz to stress-test the state-of-the-art SMT solvers Z3 and CVC4. During our testing period, we have filed numerous bugs on the issue trackers of Z3 and CVC4.

RESULT SUMMARY

- *Many bugs:* We found 322 bugs, 229 in Z3 and 93 in CVC4. Among these, 278 were already fixed.

[2] http://smtlib.cs.uiowa.edu/theories.shtml
[3] https://github.com/testsmt/yinyang

```
;;; Functions from the core theory
(not Bool Bool)
(=> Bool Bool Bool :right-assoc)
(and Bool Bool Bool :left-assoc)
(or Bool Bool Bool :left-assoc)
(xor Bool Bool Bool :left-assoc)
(par (A) (= A A Bool :chainable))
(par (A)
    (distinct A A Bool :pairwise))
(par (A) (ite Bool A A A))

;;; Functions from Ints
(- Int Int)
(- Int Int Int :left-assoc)
(+ Int Int Int :left-assoc)
(* Int Int Int :left-assoc)
(div Int Int Int :left-assoc)
(mod Int Int Int)
(abs Int Int)
(<= Int Int Bool :chainable)
(<  Int Int Bool :chainable)
(>= Int Int Bool :chainable)
(>  Int Int Bool :chainable)

;;; Functions from Reals
(- Real Real)
(- Real Real Real :left-assoc)
(+ Real Real Real :left-assoc)
(* Real Real Real :left-assoc)
(/ Real Real Real :left-assoc)
(<= Real Real Bool :chainable)
(<  Real Real Bool :chainable)
(>= Real Real Bool :chainable)
(>  Real Real Bool :chainable)

;;; Functions from Real\_Ints
(- Int Int Int :left-assoc)
(+ Int Int Int :left-assoc)
(* Int Int Int :left-assoc)
(div Int Int Int :left-assoc)
(mod Int Int Int)
(abs Int Int)
(<= Int Int Bool :chainable)
(<  Int Int Bool :chainable)
(>= Int Int Bool :chainable)
(>  Int Int Bool :chainable)
(- Real Real)
(- Real Real Real :left-assoc)
(+ Real Real Real :left-assoc)
(* Real Real Real :left-assoc)
(/ Real Real Real :left-assoc)
(<= Real Real Bool :chainable)
(<  Real Real Bool :chainable)
(>= Real Real Bool :chainable)
(>  Real Real Bool :chainable)
(to_real Int Real)
(to_int Real Int)
(is_int Real Bool)

;;; Functions from Strings
;
; Core string functions
(str.++ String String String :left-assoc)
(str.len String Int)
(str.< String String Bool :chainable)

; Regular expression functions
(str.to_re String RegLan)
(str.in_re String RegLan Bool)
(re.none RegLan)
(re.all RegLan)
(re.allchar RegLan)
(re.++ RegLan RegLan RegLan :left-assoc)
(re.union RegLan RegLan RegLan :left-assoc)
(re.inter RegLan RegLan RegLan :left-assoc)
(re.* RegLan RegLan)
(re.comp RegLan RegLan)
(re.diff RegLan RegLan RegLan :left-assoc)
(re.+ RegLan RegLan)
(re.opt RegLan RegLan)
(re.range String String RegLan)

; Misc string functions
(str.<= String String Bool :chainable)
(str.at String Int String)
(str.substr String Int Int String)
(str.prefixof String String Bool)
(str.suffixof String String Bool)
(str.contains String String Bool)
(str.indexof String String Int Int)
(str.replace String String String String)
(str.replace_all String String String String)
(str.replace_re String RegLan String String)
(str.replace_re_all
    String RegLan String String)

; Maps to and from integers
(str.is_digit String Bool)
(str.to_code String Int)
(str.from_code Int String)
(str.to_int String Int)
(str.from_int Int String)
```

FIGURE 5.4: TypeFuzz's configuration file. The syntax is purposefully adapted to the SMT-LIB theory specifications. The configuration is tailored to the theories Core, Reals, Ints, RealInts and Strings.

- *Many longstanding soundness bugs in CVC4:* We found 20 soundness bugs alone in CVC4's default mode. Many of them (7/20) are at least 2 years latent and pre-date any previous fuzzing campaign.

Evaluation Setup

We have run TypeFuzz on a machine equipped with an AMD Ryzen Threadripper 3990X with 64 cores and 32GB RAM. We occupied half of its cores. Additionally, we ran another machine equipped with an Intel Core i7-8700 CPU with 6 CPU cores of which we used full cores. Both machines were running Ubuntu 18.04 (64-bit).

TEST SEEDS & OPTIONS As the test seeds, we used non-incremental formulas from the linear and nonlinear reals and integer arithmetic, their combinations (LIA, LRA, NIA, NRA, QF_LIA, QF_LIRA, QF_LRA, QF_NIA, QF_NIRA, QF_NRA) and the string logics QF_S and QF_SLIA. All seeds were taken from the GitLab repositories provided by the SMT-LIB initiative.[4] Since their creation, the following minor modifications were made to these files: (1) README updates and satisfiability status labels, (2) removal of a few incorrectly assigned instances to QF_LIA, and (3) several updates in the QF_S and QF_SLIA seeds changing string operator labels from "`-`" to underscore, *etc.* to ensure compliance with the evolving standard. Therefore, we can safely assume that previous approaches [1, 2, 32] used the same seeds. The benchmarks range from verification of systems, proofs, synthesized programs to symbolic execution runs and randomly generated formulas. A subset of the formulas is used by the annual SMT solver competition. We mainly focused our testing efforts on the default modes of the solvers. We consider CVC4 to be in default mode, if apart from options to support SMT-LIB seeds such as `--produce-models` and `--strings-exp`, no further options are enabled. For Z3, the option `unicode=true` was necessary during the first month of the testing period to guarantee Z3's compliance with the specification of the string theory. Apart from the default mode, we focused on a few popular pre-processing options and rewriter options. These configurations are interesting since bugs in them are likely to cause undetectable soundness issues. We selected the options as per the developer's priorities. Furthermore, upon request of Z3's main developer, we have tested Z3's new core (`tactic.default_tactic=smt, sat.euf=true`), which is supposed to become Z3's default mode once stable. For CVC4, we experimented with

[4] https://smtlib.cs.uiowa.edu/benchmarks.shtml

the lazy preprocessing options `--no-strings-lazy-pp` and `--strings-lazy-pp`. For Z3, we used `rewriter.cache_all=true`, `rewriter.pull_cheap_ite=true`, `rewriter.eq2ineq=true`, `rewriter.hoist_mul=true`, and `rewriter.flat=false`.

BUG TYPES During testing, we encountered many different kinds of bugs. We distinguish them by the following categories.

- *Soundness bug*: Formula φ triggers a soundness bug if solvers S_1 and S_2 both do not crash and give different satisfiability results.
- *Invalid model bug*: Formula φ triggers an invalid model bug if the model returned by the solver does not satisfy φ.
- *Crash bug*: Formula φ triggers a crash bug if the solver throws an assertion violation or a segmentation fault.

TypeFuzz detects soundness bug triggers by comparing the standard outputs of the solvers. TypeFuzz detects invalid model bug triggers by internal errors using the model of the SMT solver. Crash bug triggers are detected whenever a solver returns a non-zero exit code and no timeout occurs.

BUG TRIGGERS Dozens of sizable bug triggers usually point to the same underlying bug. Hence, we need to de-duplicate and reduce the bug triggers. TypeFuzz collects bug triggers that may stem from the same underlying bug. Hence, we de-duplicated the bug triggers after each fuzzing run with TypeFuzz to avoid duplicate bug reports on the GitHub issue trackers. Crash bugs are either assertion violations or segmentation faults. We de-duplicate assertion violations via the location information (file name and line number) printed on standard output/error. For soundness and invalid model bugs, we first categorize the bug triggers by theory. We do this because bug triggers in different theories are likely to be unique bugs. Then, we select one bug trigger per theory at a time for reporting. If the bug was fixed, we check the remaining bug-triggering formulas of the same theory. If one of them still triggers a bug in the solver, we repeat this process until none of them triggers a bug anymore. We evaluated 897 bug trigger-seed pairs found by TypeFuzz. This number is much larger than reported bugs because a bug can often be triggered many times. The average seed size is 2,023 bytes, the average bug trigger size is 1,776 bytes. Bug triggers are in most cases not significantly larger than the seeds: 80.7% of the bug triggers are smaller than the seed, while 19.3% are larger than the seed.

Status	Z3	CVC4	Total
Reported	229	93	322
Confirmed	204	86	290
Fixed	196	82	278
Duplicate	9	7	16
Won't fix	14	0	14

(a)

Type	Z3	CVC4	Total
Crash	80	39	119
Soundness	66	30	96
Invalid model	58	17	75

(b)

#Options	Z3	CVC4	Total
default	71	47	118
1	24	27	51
2	85	11	96
3+	24	1	25

(c)

Figure 5.5: (a) Status of bugs found in Z3 and CVC4. (b) Types of the confirmed bugs. (c) # Options supplied to the solvers among the confirmed bugs.

RQ1: How effective is Generative Type-Aware Mutation?

From end of January 2021 to mid September 2021, we have extensively stress-tested the SMT solvers Z3 and CVC4 with TypeFuzz. From the 322 reported bugs, 290 were confirmed, 278 were fixed, 16 were categorized as duplicates, and 14 were won't fixes (see Figure 5.5a). As for the duplicates in Z3, their main developer followed a rather aggressive approach by categorizing every bug as duplicate for which a syntactically similar-looking formula in an open issue existed. The few won't fixes were caused by bugs which the developers confirmed. They were either viewed as not worth fixing or could not be reproduced. Among the confirmed bug (Figure 5.5b), the most frequent category are crash bugs (119 out of 290) followed by soundness bugs (96 out of 290) and invalid model bugs (75 out of 290). Most (118 out of 290) of the confirmed bugs occurred in the defaults modes of the solvers and only 51 out of 290 were with one additional option (see Figure 5.5c). For the confirmed bugs with two options in Z3, 74 out of 96 were related to the new core of Z3, which is supposed to replace Z3's default mode, once stable enough. In fact, Z3's main developer appreciated our fuzzing efforts. After dozens of bug fixes for the new core, he wrote:

> *Thanks for targeting the new code. It is a very good use of the fuzzing facilities and helps reaching a more solid state for this so-far not exercised code. All bugs reported in this thread have now been fixed.*

We also examined the logic distribution of the confirmed bugs. Most confirmed bugs in Z3 in the QF_S (40out of 204), followed by the QF_SLIA (19out of 204) and the QF_NIA (9 out of 204). For CVC4, the top-3 logics are the same: QF_S logic (39 out of 86) followed by the QF_SLIA (16 out of 86). Strikingly, TypeFuzz found 20 bugs in CVC4's default mode. Most prior

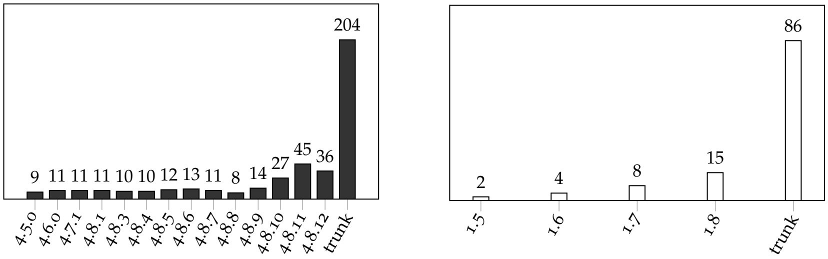


Figure 5.6: Confirmed bugs that affect releases of Z3 (left) and CVC4 (right).

approaches did not find any bugs in CVC4 [32, 63], YinYang [101] found eight in nine months and OpFuzz [2] found eleven in a year. All approaches were reportedly using the SMT-LIB seeds and similar resources as TypeFuzz did. TypeFuzz found these bugs despite the robustified Z3 and CVC4 and all the bug fixes caused by prior fuzzing campaigns.

RQ2: How significant are TypeFuzz's findings?

To understand the significance of our bug findings, we have studied the influence on historic Z3 and CVC4 releases that supported the tested logics. For CVC4, we consider all official releases versions from 1.5 (released on July 10, 2017) and later. For Z3, we consider, versions 4.5.0 (released on Nov 11, 2016) and later. Figure 5.6 shows the cumulative bug counts in the different release versions of Z3 and CVC4 respectively. In Z3, TypeFuzz found 4 bugs in the 4.5.0 release. Among these, two were invalid model bugs in QF_SLIA and QF_NIA respectively a segmentation fault in Z3's rewriter flat configuration and an assertion violation which were consecutively baked into later release versions. The first soundness bug occurs at version 4.8.8 (released on Apr 9, 2020). Two more occurred at version and nine more in 4.8.10. For CVC4, one refutational soundness bug in the default mode affects the 1.5 release, which was also baked in the 1.6 release. Two additional soundness bugs are affecting CVC4 1.6, both in the default mode. This makes 3 confirmed bugs affecting the 1.6 release.

RQ3: Are Generative Type-Aware Mutation and operator mutations orthogonal?

To answer this research question, we have run an experiment to measure the code coverage of TypeFuzz compared to its seeds, the state-of-the-art

	Z3			**CVC4**		
	lines	*functions*	*branches*	*lines*	*functions*	*branches*
Seeds	17.2%	16.5%	10.6%	21.5%	39.7%	8.0%
OpFuzz	17.8%	16.8%	11.1%	22.3%	40.9%	8.4%
TypeFuzz	19.4%	18.7%	11.9%	22.2%	40.7%	8.3%
TypeFuzz + OpFuzz	**19.7%**	**18.8%**	**12.2%**	**22.7%**	**40.9%**	**8.5%**

Figure 5.7: Line, function, and branch coverage achieved by the baseline seeds, OpFuzz, TypeFuzz and the combination of OpFuzz and TypeFuzz.

fuzzer for SMT solvers OpFuzz. We have sampled 100 files from all test seeds and then ran the following four configurations: A run on each seed from the chosen set with Z3 and CVC4 (Seeds), the state-of-the-art fuzzer OpFuzz, our tool TypeFuzz, and the sequential combination of OpFuzz and TypeFuzz, all with the initial set of seeds. The number of mutating iterations for each seed is 10 and the timeout for each solving query is 8 seconds. For all coverage measurements we used gcov [5] from the GCC suite.

Figure 5.7 shows the cumulative coverage data. We first observe that both OpFuzz and TypeFuzz cover strictly more code than the seed set on Z3 and CVC4 respectively. From the first three rows Seeds, OpFuzz, and TypeFuzz, we deduce that both OpFuzz and TypeFuzz can cover additional code as compared to the seeds. For OpFuzz, this increase is rather low (+0.6% LoC in Z3 and +0.8% LoC in CVC4) confirming previous experiments. For TypeFuzz, the increase is significantly higher in Z3 (+2.2% LoC) and slightly lower in CVC4 (+0.7% LoC). Looking again at the first three rows, we can also deduce that TypeFuzz covers code that OpFuzz does not, since TypeFuzz's percentage is higher than OpFuzz's (17.8% vs 19.4%). From the third and fourth rows, we deduce that OpFuzz also covers different code regions than Typefuzz since there is an increase in code coverage, i.e., 19.7% for TypeFuzz + OpFuzz versus 19.4% for TypeFuzz alone. This indicates that OpFuzz and TypeFuzz are complementary in terms of code coverage.

[5] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

5.5 SELECTED BUG SAMPLES

This section details multiple bug samples from our extensive bug-hunting campaign of the SMT solvers Z3 and CVC4 and inspects their root causes. The reports shown are reduced bug triggers after bug reduction.

FIGURE 5.8A shows a solution soundness bug in CVC4. The bug has existed since CVC4 1.7 was released on Apr 9, 2019 and pre-dates any fuzzing campaign. The bug is due to an inadmissible rewrite and not detected by the model validator.

```
// (= "" (str.replace x "A" "")) ---> (str.prefix x "A")
- if (StringsEntail::checkLengthOne(ne[1]) && ne[2] == empty)
+ if (StringsEntail::checkLengthOne(ne[1], true) & ne[2] == empty)
{
  Node ret = nm->mkNode(STRING_PREFIX, ne[0], ne[1]);
  return returnRewrite(node, ret, Rewrite::STR_EMP_REPL_EMP);
}
```

src/theory/strings/sequences_rewriter.cpp (132504c)

Method `bool checkLengthOne(Node s, bool strict=false)` checks whether a string expression `s` has length one. The comparison is exact if `strict=true` and otherwise requires `s` to have at most length one. The precondition for the rewrite in the above listing is to check if `str.replace`'s second argument of length one. The developer fixed the bug by enforcing the strict case.

FIGURE 5.8B shows a refutational soundness bug in Z3's QF_SLIA logic. The bug was caused by an incorrect sequence axiom. [6] The bug trigger has one assert with a negated binary equation. This format has inspired Z3's main developer to add the following rewrite along with the bugfix:

```
   indexof("", b, r) -> if b = "" and r = 0 then 0 else -1
+  indexof(a, "", x) -> if 0 <= x <= len(a) then x else - 1
   indexof(unit(x)+a, b, r+1) -> indexof(a, b, r)
```

src/ast/rewriter/seq_rewriter.cpp (e83f319)

Accordingly, Z3 will rewrite `indexof(a, "", x)` to `x` if index `x` is in the range of the string and to constant `-1` otherwise.

FIGURE 5.8C shows an invalid model in Z3's QF_SLIA logic. The formula is satisfiable but Z3 returns an invalid model on it. The bug was fixed by overhauls in the sequential rewriter of Z3.

[6] Related source file: `src/ast/rewriter/seq_axioms.cpp`

```
(declare-fun x () String)
(declare-fun y () String)
(assert (str.< x (str.replace ""
(str.++ (str.replace "B" x "")
(str.replace "B"
(str.replace "B" x "") "")) y)))
(check-sat)
```

(a) Long-latent solution soundness bug in CVC4 undetected by model validator.

https://github.com/CVC4/CVC4/issues/6075

```
(declare-fun x () String)
(declare-fun y () String)
(declare-fun z () Int)
(assert (not (= (str.substr "B" z
(str.indexof x "" (str.len x)))
(str.substr "B" z (str.len x)))))
(check-sat)
```

(b) Refutation soundness bug in Z3's QF_SLIA logic.

https://github.com/Z3Prover/z3/issues/5074

```
(declare-fun a () Bool)
(declare-fun b () Int)
(declare-fun c () String)
(declare-fun d () String)
(assert (= c (str.++ (str.replace d
(str.substr (ite a c d) 0 b) c) d)))
(check-sat)
```

(c) Invalid model bug in Z3's QF_SLIA.

https://github.com/Z3Prover/z3/issues/5140

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(assert (and (= 0 (- (div 0 0) a))
(= 0 (+ b 1 b (* c
(mod (* a (- 1)) 0))))))
(check-sat)
```

(d) Invalid model bug in Z3's QF_NIA.

https://github.com/Z3Prover/z3/issues/5136

```
(declare-fun a () Real)
(declare-fun b () Real)
(assert (= b (+ 1 (* a a
(+ 1 (/ b b))))))
(check-sat)
```

(e) Crash bug in CVC4 in QF_NRA.

https://github.com/CVC4/CVC4/issues/6228

```
(declare-fun a () String)
(assert (str.< a "ar"))
(assert (str.prefixof "ar"
(str.replace a "ar" "")))
(check-sat)
```

(f) Z3 refutation soundness bug in QF_SLIA.

https://github.com/Z3Prover/z3/issues/5117

```
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(declare-fun e () Int)
(assert (and (>= b 0) (<= b 3)
(>= c 2 d) (= c (* 2 a) d)
(= (- (- e c d)) 0) (= e 1)))
(check-sat)
```

(g) Segmentation fault in Z3 QF_LIA.

https://github.com/Z3Prover/z3/issues/5035

```
(declare-fun x () String)
(declare-fun y () String)
(assert (= (str.replace
(str.replace x "B" (str.++
"B" "B")) "B" (str.++ y "B"))
(str.++ y "B")))
(check-sat)
```

(h) Soundness bug in CVC4's QF_S logic.

https://github.com/CVC4/CVC4/issues/5915

FIGURE 5.8: Selected bug samples in Z3 and CVC4.

Figure 5.8d depicts an invalid model bug in Z3's QF_NIA logic. The formula is satisfiable but Z3 reports an invalid model on it. The issue was that Z3 did not match the integer division by zero `(div 0 0)` as an uninterpreted constant as mandated by the SMT-LIB standard. Z3's main developer fixed this regression by adding a matching case for integer division and for modulo, remainder, and division.

```
+    MATCH_BINARY(is_mod0);
+    MATCH_BINARY(is_rem0);
+    MATCH_BINARY(is_div0);
+    MATCH_BINARY(is_idiv0);
```

src/ast/arith_decl_plugin.h (c71bbb6)

Figure 5.8e presents a crash bug in CVC4 triggered by a QF_NRA formula. The pull request in response to this bug got a *major* label. According to a CVC4 developer, CVC4 was incorrectly trying to repair a model when one is not guaranteed to exist, leading to a spurious assertion failure.

Figure 5.8f shows a refutational soundness bug in z3's string logic qf_slia. the formula is satisfiable, but z3 returns `unsat` on it. a model is realized by `a = "aarr"`. The first assert is satisfied since `"aarr"` is lexicographically smaller than `"ar"`. The second assert is also satisfied by this model: if we replace the `"ar"` within `"aarr"` by the empty string, we obtain `"ar"` which is a prefix of itself. the root cause for this bug was an incorrect rewrite rule for the case when the third argument `str.replace` is empty; in z3's sequential rewriter. z3's main developer fixed this bug by removing the faulty rewrite rule and replacing it with a correct one.

Figure 5.8g depicts a segfault in Z3's `rewriter.flat=false` configuration. The segfault is caused by an issue with inconsistent assignments in the lia2pb tactic (`src/tactic/arith/lia2pb_tactic.cpp`) This issue is longstanding: it existed since Z3 4.5.0 which was released on Nov 8, 2016.

Figure 5.8h depicts a refutational soundness bug in CVC4's string logic. Similar to the bug in Figure 5.8f, the bug occurs in the sequences rewriter. The logic of the rewrite rule is detailed in the following code snippet:

```
     // (str.contains (str.replace x y z) w) --->
     //   (str.contains (str.replace x y "") w)
     // if (str.contains z w) ---> false and (str.len w) = 1
     if (StringsEntail::checkLengthOne(node[1]))
     {
-    Node ctn = d_stringsEntail.checkContains(node[1], node[0][2]);
```

```
+   Node ctn = d_stringsEntail.checkContains(node[0][2], node[1]);
```

src/theory/strings/sequences_rewriter.cpp (48047e8)

The method `bool checkContains(Node z, Node w)` decides for two string nodes whether `z` is contained in `w`. `node[0][2]` corresponds to `z` (third child of the `str.replace` expression) and `node[1]` to `w`. The bug occurred since two arguments were reversed which led to an incorrect precondition for the rewrite rule. This bug was fixed and added to the regressions.

5.6 LIMITATIONS & DATA-DRIVEN TYPE-AWARE MUTATION

Generative Type-Aware Mutation has been demonstrated to be effective for SMT solver testing. Naturally, it also comes with some limitations. First, Generative Type-Aware Mutation cannot add new assertions to the seed formula. Second, it cannot mutate unseen constants. For example, if a bug would be triggered by a term `(= (str.len x) 5)` and all but the constant "5" would occur in the seed formula, Generative Type-Aware Mutation could not generate the term and may miss the bug. Both type-aware operator mutation and FuzzChick share these limitations.

To overcome the second limitation, we experimented with another approach called *Data-driven Type-Aware Mutation*. For this technique, we use a database to store a large number of expressions along with their types in a pre-processing step. Given a seed formula φ, we then do the following:

1. Choose a random expression $expr_1$ in φ.
2. Choose a random operator op from the SMT-LIB specification of return type $type(expr_1)$
3. Build an expression $expr_2$ rooted by op using random terms from the database based on op's signature
4. Substitute $expr_1$ with $expr_2$ in φ

Using this approach, we found reported another 40 bugs, out of which 29 were confirmed, and 29 were fixed.

5.7 RELATED WORK

We first discuss related work SMT solver robustness and performance testing. Then, as Generative Type-Aware Mutation is a hybrid between mutation-based and grammar-based fuzzing, we discuss related approaches to mutation-based and grammar-based fuzzing.

SMT SOLVER ROBUSTNESS AND PERFORMANCE TESTING Our approach is particularly related to the prior works on SMT solver testing. The first approach on testing SMT solvers was the fuzzing tool FuzzSMT [60] which is based on differential testing and targets bit-vector logic. Unlike Generative Type-Aware Mutation, FuzzSMT was entirely based on grammar-based fuzzing without a mutational component. FuzzSMT totally found 16 solver defects in five older solvers, however, none in Z3. BtorMBT [61] is a testing tool for Boolector [62], an SMT solver for the bit-vector theory. BtorMBT tests Boolector by generating random, valid API call sequences. Thanks to the SMT-LIB initiative [47], SMT theories have been formalized and common input/output file formats have been devised. In addition, the yearly solver competition SMT-COMP heavily penalizes solvers with soundness issues. As a result, the SMT solvers Z3 and CVC4 have robustified and were believed to be quasi-stable. In fact, until October 2019 there were less than 50 potential soundness issues reported during eight years of development and around 150 in Z3 in 3 years [2].

Researchers have hence targeted the less mature logics such as the recently proposed theory of strings. Blotsky *et al.* [56] proposed StringFuzz which focuses on performance issues in string logic. StringFuzz generates test cases in two ways, one is mutating and transforming the benchmarks, another one is randomly generating formulas from a grammar. StringFuzz found 2 performance bugs and 1 implementation bug in z3str3. Bugariu and Müller [63] proposed a formula synthesizer for String formulas that are by construction satisfiable or unsatisfiable. They showed that their approach can detect many existing bugs in String solvers and they found 5 new soundness/incorrect model bugs in z3 and z3str3. However, it remained an open question whether automated testing tools could find bugs in theories except for the unicode string theory in Z3 and CVC4. Semantic fusion [1] is an approach to stress-test SMT solvers by fusing formula pairs that are by construction either satisfiable or unsatisfiable. Winterer, Zhang, and Su's tool YinYang found 39 bugs in Z3 and 9 in CVC4. STORM [32], another recent mutation-based SMT solver testing approach, found 27 bugs in Z3, however none in CVC4. Later, type-aware operator mutation [2] has found several hundreds of bugs in the SMT solvers Z3 and CVC4. However, recently, previous approaches have experienced the saturation effect. Generative Type-Aware Mutation has overcome the shortcomings of previous approaches by combining mutation-based and grammar-based fuzzing.TypeFuzz is a highly practical tool that SMT solver developers can use to stress-test new features conveniently.

MUTATION-BASED FUZZING Mutation-based testing fuzzing techniques leverage user-provided test seeds and generate new mutated inputs to uncover bugs in programs. The two closest works from the family of mutation-based testing techniques are skeletal program enumeration (SPE) for testing C compilers [82], and FuzzChick [83] an approach to test Coq programs. Similar to Generative Type-Aware Mutation, SPE also performs random type-aware mutations. However, in contrast to Generative Type-Aware Mutation, SPE is limited to variables and is not generative. FuzzChick generates test cases by type-aware mutation. FuzzChick stores parameter types and generates new values for the parameters while preserving type correctness. However, unlike Generative Type-Aware Mutation, FuzzChick uses coverage feedback to guide its mutations (grey-box fuzzing) while Generative Type-Aware Mutation is a black-box fuzzing technique. Grey-box fuzzing enhances black-box fuzzing by coverage information. The most prominent tool for binary grey-box fuzzing is AFL [85]. Given a set of test seeds, AFL performs mutations at the binary level, such as bit-shifts, *etc.* However, binary-level fuzzing is ineffective on programs with highly structured inputs (*e.g.* PDF viewers, programming language engines, *etc.*) because of the many syntactically invalid inputs being generated. Thus, towards structured test inputs, grammar-aware grey-box fuzzers were proposed. To generate valid test inputs, grammar-aware grey-box fuzzers were proposed. AFLSmart [88], Superion [89] and Nautilus [90] are general grammar-aware grey-box fuzzers targeting programming language engines. They use code coverage to guide the grammar-aware mutations.

GENERATIVE FUZZING Generative fuzzers [102] synthesize test inputs from scratch using a language grammar or a (language) model. Csmith [91] generates random C programs through repeated applications of rules from the C grammar. Similar to Generative Type-Aware Mutation, Csmith relies on differential testing to cross-check the generated seeds. Csmith has found 300+ bugs in the compilers GCC and LLVM. A recent follow-up work to Csmith is YarpGen [103] which additionally prevents generating C programs with undefined behavior. Another recent generative fuzzing approach is pivot query synthesis (PQS) [104] It synthesizes specific SQL queries on random databases. Unlike Csmith and YarpGen, PQS is a metamorphic testing approach. Moreover, researchers have adapted generative language models [105] to generate and guide input generation. Different from all other approaches, Generative Type-Aware Mutation does not generate from scratch but generates through the substitution of expressions.

6

JANUS: FINDING INCOMPLETENESS BUGS VIA WEAKENING AND STRENGTHENING

This chapter focuses on incompleteness bugs in SMT solvers. We first formally define incompleteness bugs. Then, we propose Weakening and Strengthening, an approach for fabricating an implication chain of formulas. Next, we present Janus, its realization, and show that it is capable of finding 31 incompleteness bugs in Z3 and cvc5, out of which 26 were confirmed, and 20 were fixed. Finally, we discuss Janus's incompleteness bug findings revealing functional, regression, and performance issues in the solvers—several of which triggered in-depth discussions among the developers.

6.1 Motivation

An SMT solver returns `unknown` if it cannot cannot determine the satisfiability of a formula. Incompleteness bugs, *i.e.* unexpected `unknown`-results, impact the performance of SMT solvers' client applications frustrating their developers—especially since SMT solvers are usually at the very core of their client software solving NP-hard problems. Formula φ may realize a path constraint in a symbolic execution engine (*e.g.* KLEE [9], Microsoft's SAGE [52]), an access policy of a web service (*e.g.* AWS's Zelkova [16]), or a model of a safety-critical system (*e.g.* AdaCores's Spark [17]). Potential consequences of incompleteness bugs include missed bugs in the software under test, slow (or even non-terminating) verification of safety-critical or security-critical properties and other undesirable effects. The following trace shows an incompleteness bug in Z3:

```
$ z3 bug.smt2
unknown

$ cat bug.smt2
(declare-const x Int)
(assert (forall ((v Int)) (= v (* x x))))
(check-sat)
```

The formula asserts that every integer can be represented as the square of some other integer. The first statement of the script declares an integer variable, the second specifies a constraint, and the third queries the SMT solver. Since we cannot choose an admissible value for `x` to satisfy the constraint (as there is no square number equal to all integers), the formula is unsatisfiable, and Z3 should return `unsat`. If the formula was a proof or a path condition such behavior may lead to verification failure. Why does Z3 fail at solving such a simple formula? As it turns out, it is caused by a bug in the implementation of model-based quantifier instantiation (MBQI). MBQI guesses values for the universal quantifiers (`v` in our case) to check whether the formula is unsatisfiable. On inspecting the issue deeper, we noticed that even if we run MBQI a million iterations, Z3 could not decide the formula. [1] However, fixing a random integer `v` which is not a square number would have been enough to determine the unsatisfiability of the formula. We reported this bug to the issue tracker of Z3 on GitHub. Z3's main developer promptly fixed it.

Not every `unknown`-result indicates a bug. As SMT solvers support undecidable logics, they are necessarily incomplete. An SMT solver returns `unknown` on a formula if it has no decision procedure to solve the formula or to avoid a timeout. In practice, SMT solver can solve most problem instances from undecidable logics relevant to users. Similar to decidable logics, SMT solver developers enhance their solvers by rewriter rules, pre-processors *etc.* However, distinguishing expected from unexpected incompletenesses is difficult and confuses users:

> *"I'm seeing a regression [...], where a lot of simple formulas that used to be unsat now give unknown."*
>
> https://github.com/Z3Prover/z3/issues/5516

> *"The following code will produce unsat in z3 version 4.8.10.0 but is unknown in later versions."*
>
> https://github.com/Z3Prover/z3/issues/5438

> *"This is pretty unexpected since the query is small and does not contain features where we would expect to see performance regressions when updating releases."*
>
> https://github.com/Z3Prover/z3/issues/4702

[1] `z3 smt.mbqi.max_iterations=1000000 bug.smt2`

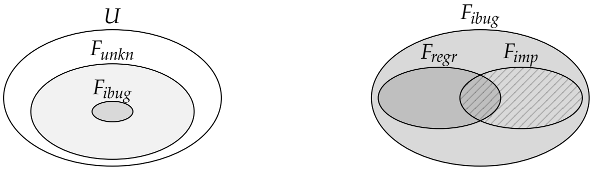


FIGURE 6.1: Classification of incompleteness bugs: Formulas F_{regr} and F_{imp} are the targets for incompleteness testing.

The first comment is from a developer of LLVM static analyzer Alive2 [106], the second from Haskell verifier SBV [107], and the third issue is a query of software verifier SMACK [108]. Incompleteness bugs are not only a Z3 issue. On CVC5's issue tracker, users report similar experiences. [2] SMT solver developers address such issues by ad-hoc fixes, pointing to the logic's undecidability and suggesting workarounds. Such hacks can lead to bugs masked as unknowns. This might lead to incorrect conclusions, suggesting that the solver returns `unknown` because the problem is undecidable.

6.2 PROBLEM STATEMENT

We consider the problem of finding incompleteness bugs F_{ibug} in an SMT solver, *i.e.*, unexpected incompletenesses among the set of inputs F_{unkn} on which the SMT solver returns `unknown`. We approximate F_{ibug} as follows.

Definition 6.2.1 (Incompleteness bugs in SMT solvers)**.** We distinguish the following two types of incompletenesses bugs:

1. **Regression incompleteness**:
   $S_{old}(\varphi) = \texttt{sat/unsat}$ and $S(\varphi) = \texttt{unknown}$

2. **Implication incompleteness**:
   $S(\varphi) = \texttt{sat}$ and $S(\varphi') = \texttt{unknown}$ if φ implies φ'
   $S(\varphi) = \texttt{unsat}$ and $S(\varphi') = \texttt{unknown}$ if φ' implies φ

where S is an SMT solver, S_{old} is an earlier version of S, φ is an SMT formula and φ' is a mutated formula based on φ.

Figure 6.1 classifies incompleteness bugs: On the left, we see the "Universe" U of all SMT formulas handled by the solver. The set U contains

[2] https://github.com/cvc5/cvc5/issues/6274

```
$ z3 5338.smt2
unknown

$ z3-4.8.10 5338.smt2
unsat

$ cat 5338.smt2
(assert (forall ((v Int)) (= 0 v)))
(assert (= 0 (mod 0 0)))
(check-sat)
```

(a)

```
$ cvc5 -q 7009-mut.smt2
sat

$ cat 7009-mut.smt2
(declare-fun s () Real)
(declare-fun k () Real)
(assert (= (* s k) 1))
(check-sat)
```

```
$ cvc5 -q 7009.smt2
unknown

$ cat 7009.smt2
(declare-fun s () Real)
(declare-fun k () Real)
(assert (>= (* s k) 1))
(check-sat)
```

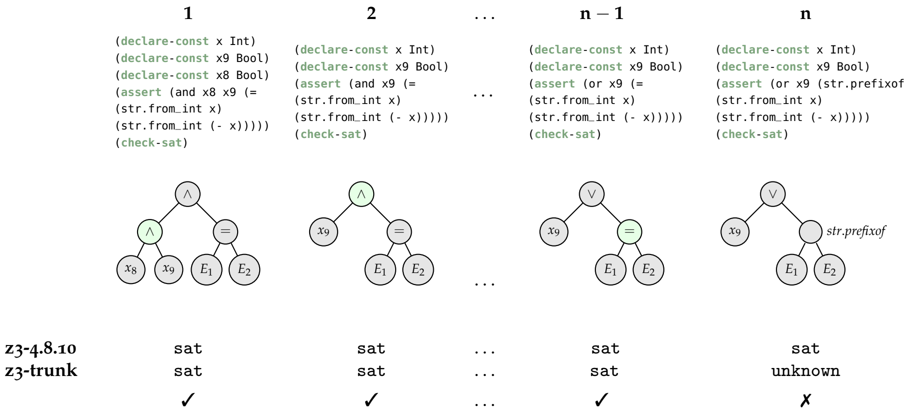
(b)

FIGURE 6.2: (a) Regression incompleteness: legacy Z3 4.8.10 solves this formula, while the trunk returned `unknown` (Z3#5338). (b) Implication incompleteness in cvc5 caused by minor operator change (cvc5#7009).

formulas for which the SMT solver returns `unknown` (F_{unkn}) which in turn contains the set of incompleteness bugs (F_{ibug}). On the right, we have F_{ibug} which contains regression incompletenesses F_{regr} and implication incompletenesses F_{imp}. Regression incompletenesses are caused by (recent) code changes leading to an incompleteness on previously solved formulas. Typically such bugs affect client software that works correctly with an older version of the SMT solver but fails after updating the solver.

As an example, consider Figure 6.2a where Z3's trunk version does not solve a simple formula and returns `unknown` however legacy version Z3 4.8.10 determines the formula to be `unsat`. We reported this issue on the issue tracker of Z3. It was fixed within a week by the Z3 developers.

Implication incompletenesses occur when an SMT solver solves a given input formula φ but minor changes in the formula (*i.e.* the mutation to φ') cause the solver to report `unknown`. Such formula pairs can suggest possible improvements for SMT solvers, *e.g.*, to formula rewriters, pre-processors, theory solvers *etc.* If φ was generated by a client application of an SMT solver, fixing implication incompletenesses makes the client application more robust. For an example of an implication incompleteness, consider Figure 6.2b where the operator change = to >= caused cvc5 to be incomplete. Both bugs are real cases found by our approach Janus and fixed by the SMT solver developers of Z3 and cvc5.



1

```
(declare-const x Int)
(declare-const x9 Bool)
(declare-const x8 Bool)
(assert (and x8 x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

2

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (and x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

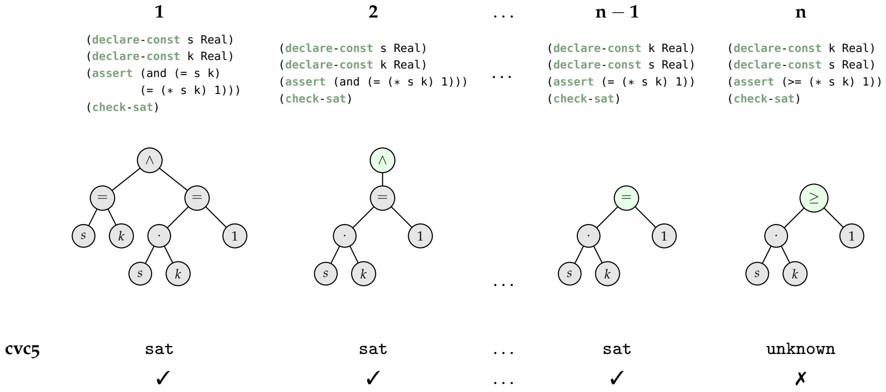
n − 1

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (or x9 (=
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

n

```
(declare-const x Int)
(declare-const x9 Bool)
(assert (or x9 (str.prefixof
(str.from_int x)
(str.from_int (- x)))))
(check-sat)
```

FIGURE 6.3: Finding regression incompleteness with Janus: sample mutation chain leading to a regression incompleteness in Z3 (Z3#5381).



1

```
(declare-const s Real)
(declare-const k Real)
(assert (and (= s k)
         (= (* s k) 1)))
(check-sat)
```

2

```
(declare-const s Real)
(declare-const k Real)
(assert (and (= (* s k) 1)))
(check-sat)
```

n − 1

```
(declare-const k Real)
(declare-const s Real)
(assert (= (* s k) 1))
(check-sat)
```

n

```
(declare-const k Real)
(declare-const s Real)
(assert (>= (* s k) 1))
(check-sat)
```

FIGURE 6.4: Finding an implication incompleteness with Janus: sample mutation chain leading to an implication incompleteness in cvc5 (cvc5#10891).

6.3 THE JANUS FRAMEWORK FOR FINDING INCOMPLETENESS BUGS

This section presents Janus, our approach to tackle regression and implication incompletenesses. We present (1) an approach overview, (2) present the necessary background, and (3) Weakening and Strengthening, the core technique of the Janus framework.

6.3.1 *Approach Overview*

Janus has two concurrent modes, one for finding regression incompletenesses, and another one for finding implication incompletenesses. We will describe them in two separate examples.

FINDING REGRESSION INCOMPLETENESSES Figure 6.3 shows a mutation chain of Janus for finding regression incompletenesses (from left to right). Janus starts with a seed formula on which both Z3 and legacy z3-4.8.10 return sat (step 1). Janus then chooses a *random* rule from its rule set and applies it to the seed (step 2). The process continues up to the point where Z3 returns `unknown` and z3-4.8.10 returns `sat`. Janus detected a regression incompleteness. This is real case, *i.e.* an actual bug that we reported to the issue tracker of Z3.

FINDING IMPLICATION INCOMPLETENESSES Figure 6.4 shows a mutation chain of Janus for finding regression incompletenesses (from left to right). Janus starts with a satisfiable seed formula (step 1). Janus then chose a satisfiability-preserving transformation rule, *e.g.*, dropping the first conjunct of the `and` expression. As the oracle is `sat`, we weaken, if the oracle was `unsat`, we would strengthen. This results in a mutated formula (step 2) satisfiable by construction. Janus generates mutants this way until the solver returns `unknown`. SMT solver developers can investigate the unknown case together with the rule (c.f. step $n-1$ to n) that led to the `unknown`-result to understand why the SMT solver has failed. This is a real case, *i.e.* an actual bug that we reported to the issue tracker of cvc5.

6.3.2 *Weakening & Strengthening*

This section presents Weakening and Strengthening, our approach to tackle regression and implication incompletenesses in SMT solvers. We first define the notion of weaker/stronger for formulas and introduce parity.

DEFINITIONS We use standard notions of typed higher-order logic, such as term, quantifier, function, *etc.*, and write expressions for term occurrences. The set of *free variables* in a formula φ is denoted as $FV(\varphi)$. We define a *subformula* to be a predicate represented by a subtree of the abstract syntax tree of φ. Departing from standard notation, we distinguish between multiple occurrences of syntactically equal parts within a formula, *i.e.*, $\varphi[F \mapsto G]$ is the formula represented by an abstract syntax tree of an SMT program where F is replaced by G in exactly one place. We write the universal and existential quantification of a term `t` over a variable `x` with sort `T` as `(forall ((x T)) t)`.

Definition 6.3.1 (Weaker/stronger)**.** Let φ_1, φ_2 be formulas with the same free variables *i.e.*, $FV(\varphi_1) = FV(\varphi_2) = \{x_1, \cdots, x_n\}$. We call φ_1 *weaker* than φ_2 if $\forall x_1, \ldots, x_n : \varphi_2 \rightarrow \varphi_1$. Conversely, we call φ_2 stronger than φ_1.

Definition 6.3.2 (Parity)**.** For a formula φ with a subformula F, we define $parity(F, \varphi)$ as

$$parity(F,\varphi) := \begin{cases} 1 & \text{if } F \text{ represents } \varphi \\ -1 \cdot parity(F,\varphi') & \text{if } \varphi = \neg\varphi' \\ parity(F,\varphi_1) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ and } F \text{ in } \varphi_1 \\ parity(F,\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ and } F \text{ in } \varphi_2 \\ parity(F,\varphi') & \text{if } \varphi = \exists x \colon \varphi' \end{cases}$$

If $parity(F, \varphi) = 1$, then F is called *positive* and *negative* otherwise.

Parity links local weakening and strengthening of a subformula to global weakening and strengthening of the overall formula. More precisely, the parity of a subformula captures whether weakening or strengthening has the same or the opposite effect on the surrounding formula.

Lemma 6.3.1. *Let φ be a formula with a subformula F. For any G weaker than F, we have:*

$$\begin{array}{ll} \textit{if } F \textit{ positive in } \varphi \textit{ then} & \forall x_1, \ldots, x_n \colon \varphi \rightarrow \varphi[F \mapsto G] \\ \textit{if } F \textit{ negative in } \varphi \textit{ then} & \forall x_1, \ldots, x_n \colon \varphi[F \mapsto G] \rightarrow \varphi \end{array}$$

with the set of free variables $FV(\varphi) = x_1, \ldots, x_n$.

For the proof of Lemma 6.3.1, we refer the reader to the Appendix A.3. Given Lemma 6.3.1, we can derive four weakening/strengthening rules.

Theorem 6.3.1. *Let F, F_w, F_s be formulas and φ a sentence such that F is a subformula of φ, F_w is weaker than F and F_s is stronger than F. Then the following statements hold:*

1. *If F is positive and φ satisfiable, then $\varphi[F \mapsto F_w]$ is satisfiable*
2. *If F is negative and φ satisfiable, then $\varphi[F \mapsto F_s]$ is satisfiable*
3. *If F is negative and φ unsatisfiable, then $\varphi[F \mapsto F_w]$ is unsatisfiable*
4. *If F is positive and φ unsatisfiable, then $\varphi[F \mapsto F_s]$ is unsatisfiable*

Proof. Cases (1) - (4) are direct corollaries of Lemma 6.3.1. □

MUTATION RULES A rule consists of two patterns, the left and right-hand side of an implication or equivalence. Many of the rules are parametrized over additional terms, e.g. $t_1 = t_2$ of Reals is equivalent to $t_1 + c = t_2 + c$ for any term c of sort Real. We instantiate such parameters in a two-stage process: (1) search the current SMT-file for terms of the required sort. If there are any, choose one randomly. Otherwise, (2) choose randomly from a set of literals. Rules can be implemented from left to right, right to left or both but we will omit this detail and present them only on the logical level. Our implemented rule set is shown in Table 6.1. Multiple rules are equivalences, e.g., the rule for "or", and implication "→". We include such rules as they diversify the set of generated mutants. They can help trigger other rules to become applicable. Janus applies "equivalence rules" in both directions to avoid getting stuck.

INTUITION BEHIND WEAKENING AND STRENGTHENING Weakening and Strengthening's key goal is verifying the robustness of SMT solvers concerning their completeness. Users expect small changes on a decidable formula φ to yield a decidable mutated formula φ'. If an SMT solver returns `unknown` on φ', it can indicate either an incompleteness bug or an expected incompleteness. Weakening a satisfiable formula (φ to a formula φ') relaxes φ's constraints. Hence, φ' admits more solutions than φ. Solving φ' is expected to be easier than solving φ. Strengthening an unsatisfiable formula (φ to a formula φ') tightens φ's constraints making it even more obvious to the SMT solver that φ' should be unsatisfiable. We view SMT solvers as black-boxes without any assumptions about the decision procedures used for a given formula. There is also no guarantee for the mutated formulas φ' to be easier to solve than φ. However, this holds on average, as the following experiment shows. We sampled 1,000 nonlinear benchmarks (500 satisfiable,

Type	Strong	Weak	Legend
Real/Int			$n_1, n_2 \in \mathbb{R}$ or $n_1, n_2 \in \mathbb{N}$
	$n_1 = n_2$	$n_1 \geq n_2 \mid n_1 \leq n_2$	
	$n_1 > n_2$	$n_1 \geq n_2 \mid n_1 \neq n_2$	
	$n_1 < n_2$	$n_1 \leq n_2 \mid n_1 \neq n_2$	
	$n_1 \odot n_2$	$(n_1 + c) \odot (n_2 + c) \mid n_1 \odot (n_2 + c) \mid n_1 \odot (n_2 + c)$	$c \in \mathbb{N}$ or $c \in \mathbb{R}$ $\odot \in \{=, >, <, \leq, \geq\}$
Bool			$\varphi, \varphi_1, \varphi_2$ boolean formulas
	$\varphi_1 \wedge \varphi_2$	$\varphi_1 \vee \varphi_2 \mid \varphi_1$	
	$\varphi_1 \oplus \varphi_2$	$\varphi_1 \vee \varphi_2$	$\oplus$ is the logical xor
	$\forall x\colon \varphi \mid \varphi[x \mapsto B]$	$\exists x\colon \varphi$	B expression of type of x
	$x_1 = \ldots = x_n$	$f(x_1) = \ldots = f(x_2)$	$x_1, \cdots, x_n$ are terms
	$\varphi_1 \vee \varphi_2$	$\exists b\colon ite(b, \varphi_1, \varphi_2)$	*ite* is if-than-else b is a boolean variable
	$ite(B, \varphi_1, \varphi_2)$	$B \rightarrow \varphi_1 \mid \neg B \rightarrow \varphi_2$	
	$\varphi_1 \rightarrow \varphi_2$	$ite(\varphi_1, \varphi_2, \top) \mid ite(\neg\varphi_1, \top, \varphi_2) \mid \forall b\colon (\varphi_1 \wedge b \rightarrow \varphi_2 \wedge b)$	
	$\varphi_1 \vee \varphi_2$	$\neg\varphi_1 \rightarrow \varphi_2$	
	$\forall x.\varphi$	$\varphi[x \mapsto B]$	
	φ_1	$\varphi_1 \vee \varphi_2$	
String			s_1, s_2, s_3 are strings
	$s_1 = s_2$	$s_1 \neq (s_1 \mathbin{++} s_3) \mid s_1 \leq_s s_2 \mid prefixof(s_1, s_2) \wedge suffixof(s_1, s_2) \mid prefixof(s_1, s_2) \wedge prefixof(s_2, s_1) \mid contains(s_1, s_2) \mid suffixof(s_1, s_2) \wedge suffixof(s_2, s_1) \mid prefixof(s_1, s_2) \mid suffixof(s_1, s_2)$	$++$: string concatenation
	$s_1 <_s s_2$	$s_1 \neq s_2 \mid s_1 \leq_s s_2$	$\leq_s$: lexicographical ordering
	$s_1 \leq_s s_2$	$substr(s_1, 0, ite(0 \leq i \leq len(s_1) - 1, i, len(s_1))) \leq_s s_2 \mid s_1 \leq_s (s_2 \mathbin{++} s_3)$	
	$contains(s_1, s_2)$	$len(s_1) \geq len(s_2)$	
Regex			$r, r_1, \cdots, r_n$ regexes, $n \in \mathbb{N}$
	r	r^+	$+$: Kleene plus
	r	$loop(1, n, r)$	$loop(i, n, r) = \bigcup_i L(r)^i$
	r	$opt(r)$	$opt(r) := union(r, (\text{str.to\_re ""}))$
	$r_1 + +_r \ldots + +_r r_n$	$union(r_1, \ldots, r_n)^n$	$++_r$: regex concat
	r	$\forall x\colon union(r, s)$	for an arbitrary string s
	r^+	r^*	$*$: Kleene star
	$range(s_1, s_2)$	$range(s_3, s_4)$	for strings s_1, s_2 strings s_3, s_4 s.t. $range(s_1, s_2) \subseteq range(s_3, s_4)$

TABLE 6.1: Weakening and strengthening rules for core logic, reals and integers, strings, and regexes. Symmetric cases are omitted for brevity. The legend column describes newly introduced symbols per group.

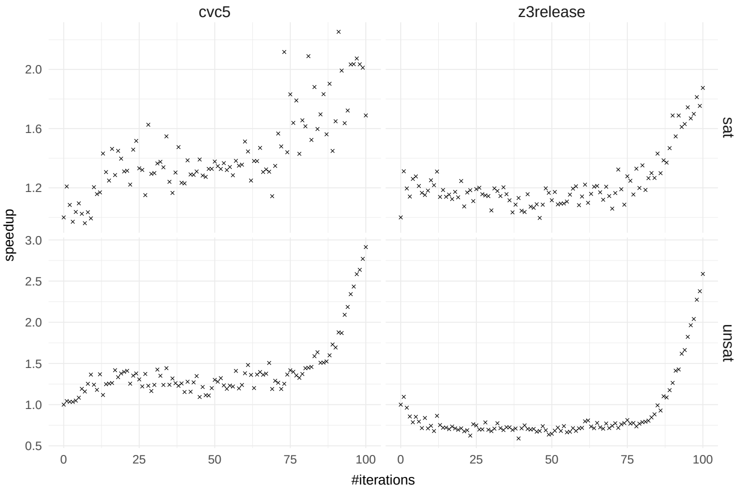


Figure 6.5: Z3 and cvc5 runtime performance averages on 1,000 evenly distributed sat/unsat nonlinear arithmetic benchmarks per weakening/strengthening iteration (1-100). Timeout: 8 seconds, files on which timeouts occurred excluded.

500 unsatisfiable) and measured Z3 and cvc5's runtime performance, after 100 weakening and strengthening steps and measured the average speedup. Figure 6.5 shows the results of this experiment.

JANUS'S IMPLEMENTATION. We built Janus on top of the SMT solver testing framework YinYang [101] in 1.5k lines of Python code. Our implementation first parses and type checks a seed formula before incrementally applying randomly selected mutation rules. After a fixed number of mutations, we restart the mutation chain from the seed. Type checking allows us to apply rules only when they are applicable and choose random terms of the correct type. At each mutation step, we forward the mutants to the SMT solvers to test for regression and implication incompleteness.

6.4 EVALUATION

From June 2021 - August 2021, we conducted a fuzzing campaign with Janus. We deployed Janus on an AMD Ryzen Threadripper 3990X 64-Core Processor with 256GB of RAM running Ubuntu 18.04. We experimented with the configuration of the fuzzer and a typical instance used 300 iterations per seed, 25 incremental mutations before resetting the seed and a solver timeout of 10 seconds. We tested the two state-of-the-art SMT solvers Z3 and cvc5. We daily rebuilt the trunk versions of Z3 and cvc5 respectively, and tested them in their default modes, *i.e.*, without any additional options besides `--strings-exp` for cvc5 to enable support for string logic. As seed files, we used SMT-LIB benchmarks from the YinYang project which are categorized into satisfiable and unsatisfiable instances. [3]

BUG TRIGGER REDUCTION We reduce bug triggers of both regression and implication incompletenesses with the SMT-specific test-case reducer ddsmt [81]. The tool repeatedly shrinks the bug-triggering formula while maintaining an invariant specified by a user-defined script called the *interestingness test*. The interestingness test is executed after each shrinking operation of ddsmt returning exit code 0 if the shrinking operation was admissible and 1 otherwise. Based on this feedback, ddsmt shrinks the bug-triggering formula to a locally minimal size, usually small enough for reporting on the issue trackers of Z3 and cvc5.

For regressions, where one solver reports `sat`/`unsat` on the bug-triggering formulas and another one reports unknown `unknown`, we specify the interestingness test by string matching the `unknown` of the second solver. Reducing implication incompletenesses is more challenging as we have to keep two related formulas in sync. Algorithm 7 shows a procedure realizing the interestingness test for implication incompletenesses. The procedure takes the bug-triggering formula φ, the rule m, and an SMT solver S as its input. We first solve φ with S (line 2). If S decides φ, we proceed; otherwise, we exit with 1 indicating that φ is not interesting (line 10). We retrieve all subformulas from φ on which m is applicable in the set *candidates* (line 4). We apply m to each subformula c, and obtain ψ (line 6). We check whether S returns `unknown` on it (line 9) and if so, we return 0 indicating that formula φ is an interesting input, *i.e.*, triggers the bug.

BUG TRIGGER SELECTION In our experiments, fuzzing for incompleteness bugs with Janus resulted in too many bug triggers to report directly to the issue trackers of the solvers. We hence manually selected the most interesting cases. One source of cases is the fact that solvers typically implement the full SMT-LIB standard in parsing, but their reasoning engines only support a subset. We filter out formulas with such unsupported features using

[3] https://github.com/testsmt/semantic-fusion-seeds

Algorithm 7 Interestingness test for implication incompletenesses

```
procedure INTERESTINGNESS_IMPLICATION($\varphi$, $m$, $S$)
    $r_1 \leftarrow S(\varphi)$
    if $r_1 \in \{$sat, unsat$\}$ then
        $candidates \leftarrow \{e \in expr(\varphi) \mid m$ is applicable to $e\}$
        for $c$ in $candidates$ do
            $\psi \leftarrow$ apply $m$ to $c$ in $\varphi$
            $r_2 \leftarrow S(\psi)$
            if $r_2 =$ unknown then
                return 0
    return 1
```

basic text search tools, as they trivially trigger an unknown response and provide no new insights to the developers. We continuously adapted our selection process to the developer feedback and solver-specific behaviors. The developers informed us of language features that are not expected to be supported well or how specific logic solvers and options should be used to validate incompletenesses.

bug findings Figure 6.6a shows the results of our bug-hunting campaign with Janus. We have totally reported 31 incompleteness bugs, 13 in Z3 and 18 in cvc5. Out of these, 26 bugs got confirmed and 20 bugs got fixed. There are 8 fixes in Z3, and 12 in cvc5 got fixed. We can partially explain this by the different development styles of Z3 and cvc5. In Z3, many incompleteness bugs were fixed on the spot by Z3's main developer with the fix being promptly pushed to Z3's master branch. In cvc5, on the other hand, several developers discuss issues, file pull requests, *etc.* Hence several of our reports are still in the queue waiting to be merged to cvc5's master branch. Among the confirmed bugs, we found 19 regression incompleteness bugs and 7 implication incompleteness bugs (see Figure 6.6b). We found more regression bugs since the reduction process is faster. Out of the 31 reported bugs, 5 regression incompletenesses in Z3 were categorized as rejected. Z3's main developer considered two of them acceptable incompletenesses in Z3. Another one was rejected by Z3's main developer since the bug did not trigger in Z3's new core. Z3's new core is an experimental configuration, intended at replacing the Z3's current core in the future. Hence, Z3's main developer saw no benefit in fixing this issue in Z3's current core. Another

Status	Z3	cvc5	Total
Reported	13	18	31
Confirmed	8	18	26
Fixed	8	12	20
Rejected	5	0	5

(a)

Type	Z3	cvc5	Total
Regression	7	12	19
Implication	1	6	7

(b)

FIGURE 6.6: (a) Statuses of incompleteness bug reports, (b) types among the confirmed incompleteness bug reports.

bug was rejected by Z3's main developers since fixing it would interfere with axioms of the string solvers. Yet another one was deemed a won't fix and disappeared after a recent code change in Z3.

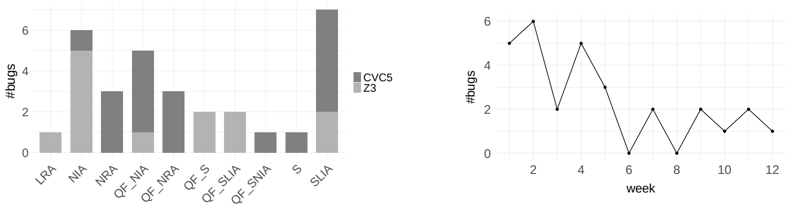


FIGURE 6.7: Left: Bug logic distribution of the reported bugs. Right: Bug reports from the start to the end of the bug-hunting campaign.

LOGICS DISTRIBUTION AMONG THE BUGS Consider Figure 6.7 left for an overview of the distribution among the reported bugs. Among the reported bugs, we found 7 bugs in SLIA, 6 in NIA, 5 in QF_NRA, 3 in NRA, 2 in QF_S, 2 QF_SLIA, and 1 in QF_SNIA, S and LRA, respectively.

DURATION OF THE CAMPAIGN & STATISTICS The bug findings are evenly distributed for the fuzzing campaign (see Figure 6.7 right). In the last two weeks, we, in fact, withheld some of our bug findings to give the developers time to fix the pending bugs from earlier weeks. During the campaign from June 2021 to Aug 2021 Janus totally generated 106 million tests, among which 760k were unknown results. Most of these were duplicates; after manual bug trigger selection, 426 remained. Since this was still quite a large number to be reported, we were conservative in reporting them to the issue trackers of Z3 and cvc5. We then reported 31 bugs to the issue trackers of the solvers for the developers to inspect.

6.5 SELECTED BUG SAMPLES

This section analyzes exemplary bug reports to illustrate the diverse incompleteness bugs that Janus can find, all depicted in Figure 6.8.

FAULTY MBQI IMPLEMENTATION IN Z3 (FIGURE 6.8A). The formula shows a bug in Z3's of model-based quantifier instantiation (MBQI), a procedure for quantifier elimination. For the reported formula, MBQI repeatedly guesses values for universally quantified variables, *i.e.*, x in our case. However, Z3 fails to solve this simple formula, which is unexpected as many values for x would satisfy the formula. Interpreting the semantics of the "Is there a square number equal to all integers?" lets us decide the formula to be unsatisfiable. A deeper analysis revealed that if we massively increase MBQI's iteration cutoff to one million, Z3 could still not decide the formula. Z3's main developer fixed this bug, the cause was an uninitialized variable.

BUG IN CVC5 REWRITE PRECEDENCE RULES (FIGURE 6.8B). Intuitively, the formula is easily satisfiable by setting variable T to true in the first assert. However, cvc5 returns unknown on the formula. The issue was detected as a regression, stable legacy releases cvc5 1.8 and 1.7 can decide the formula. Thanks to our report, cvc5 developers discovered that a set of newer rewrite rules were taking precedence over older rewrites. A cvc5 developer described this as follows:

> *"Commit 11c1fba added new rewrites for ITE. Due to the new rewrites taking precedence over existing rewrites, it could happen that some of the previous rewrites did not apply anymore even though they would have further simplified the ITE."*

The bug was roughly one-year latent which the referenced commit indicates. It was undetected by the ongoing SMT solver fuzzing campaigns and cvc5's test suites. The developers fixed this bug by adjusting the precedence rules in the rewriter.

COMPLETENESS REGRESSION IN Z3'S LRA LOGIC (FIGURE 6.8C). The formula belongs to the essentially propositional logic which is decidable. The formula contains a single quantifier over a boolean variable which could be eliminated by grounding (setting x to true

```
(declare-const x Int)
(assert (forall
        ((v Int)) (= v (* x x))))
(check-sat)
```

(a) Incompleteness bug in Z3 caused by faulty MBQI implementation.

https://github.com/Z3Prover/z3/issues/5376

```
(declare-const T Bool)
(declare-const v String)
(assert (ite T T true))
(assert (or T (and (str.prefixof v "")
(exists ((x Int)) (= "t"
(str.substr v 0 x))))))
(check-sat)
```

(b) Incompleteness in cvc5 caused by faulty rewrite precedence rules.

https://github.com/cvc5/cvc5/issues/6717

```
(declare-const x2 Bool)
(declare-const x9 Bool)
(declare-fun x () Real)
(assert (< x (ite (forall
((x Bool)) (ite x x9 x2)) 0.0 1.0)))
(check-sat)
```

(c) Regression in Z3 in a decidable logic (booleans + linear real arithmetic)

https://github.com/Z3Prover/z3/issues/5340

```
(declare-const P String)
(declare-const T String)
(assert (=
(str.is_digit T)
(str.is_digit P)))
(check-sat)
```

(d) Bug on string formula in Z3 caused by unhandled `str.is_digit`.

https://github.com/Z3Prover/z3/issues/5491

```
(declare-const a Bool)
(declare-fun b () Real)
(assert (or a (= 0
        (* b b b))))
(assert (> (* b b) 3))
(check-sat)
```

(e) Completeness regression in cvc5's QF_NRA logic.

https://github.com/cvc5/cvc5/issues/6798

```
(declare-const u Bool)
(declare-fun v () String)
(assert (exists ((x Int))
(or (not (>= 0 (str.len
(str.substr (str.replace_re ""
(str.to_re v) v) x 1)))) u)))
(check-sat)
```

(f) Regression with existential quantifier elimination.

https://github.com/cvc5/cvc5/issues/6727

```
(declare-const x Int)
(declare-fun T () Int)
(declare-fun va () String)
(assert (distinct (str.from_int T)
(str.replace va (str.replace ""
va "") (str.from_int (- x)))))
(check-sat)
```

(g) Regression bug in Z3 which led to new rewrite being implemented.

https://github.com/Z3Prover/z3/issues/5399

```
(declare-const s Real)
(assert (or (or false (= 0.0 s))
(< (* s (+ 6 (* s 12))) (- 1))))
(check-sat)
```

(h) Order sensitivity (`or`) which raised a discussions among the developers.

https://github.com/cvc5/cvc5-projects/issues/279

Figure 6.8: Selected bug samples in Z3 and cvc5.

and false). Hence, we would expect SMT solvers to decide the formula. However, Z3 returns `unknown` on it. Z3's main developer fixed the bug by refining the default tactic of Z3.

INCOMPLETENESS BUG ON STRING FORMULA IN Z3 (FIGURE 6.8D). The formula equates two `str.is_digit` expressions, each with a free string variable as a single argument. This formula should be sat. However, Z3 could not decide it, returning `unknown` since Z3 did not handle `str.is_digit` although it can decide other string formulas with such expressions. The bug was detected as an implication incompleteness and was promptly fixed by Z3's main developer.

COMPLETENESS REGRESSION IN CVC5'S QF_NRA LOGIC (FIGURE 6.8E). The formula shows a completeness regression in cvc5 on a formula that legacy cvc5 1.7 could still decide. The bug was confirmed by a cvc5 developer. and was marked with the label "bug" on the cvc5 issue tracker reflecting its high priority.

REGRESSION WITH QUANTIFIER ELIMINATION IN CVC5 (FIGURE 6.8F). The formula contains a single existential quantifier and could not be solved by cvc5. If the existentially quantified variable is however removed, cvc5 can decide the formula. The developer's feedback was that they will consider enabling pre-skolemization by default, i.e. compiling away existential quantifiers. He delegated this issue to one of his fellow developers for the fixing.

DISCOVERING A NEW STR.REPLACE REWRITE IN Z3 (FIGURE 6.8G) This formula contains the term `(str.replace "" va "")` which is equivalent to the empty string `""`. Z3 returned `unknown` for the shown formula, but correctly solved it as `sat` after we manually performed this rewrite step. To fix the issue, the developers added this exact rewrite step to the string rewriter component of the solver.

ORDER SENSITIVITY OF OR IN CVC5 (FIGURE 6.8H). This case showed the following behavior: The solver result changes from `sat` to `unknown` when replacing a subformula `f` with the equivalent `(or false f)`. From a user's perspective, this is surprising behavior and one might expect the solver to be robust against such minor simplifications. Indeed, the solver does simplify `(or false f)` to `f` but this changes the internal order of disjuncts which triggers the `unknown`.

> *"The underlying reason is [...] due to cvc5 being sensitive to the order of the terms."*
>
> *"[...] Indeed, false is removed by rewriting as one would expect, and the unknown is due to the ordering."*

After careful analysis, the developers solved deemed incompleteness as an acceptable artifact of the solver's internals. Nevertheless, they kept the issue in a collection of challenges.

6.6 RELATED WORK

Our approach is particularly related to the prior works on SMT solver robustness testing [1, 2, 30, 32, 63]. Closely related is Bugariu and Müller's approach [63]. Bugariu and Müller's formula rule-based synthesizer generates formulas that are by construction satisfiable or unsatisfiable. However, different from our approach they use equivalence-preserving rules and their approach is limited to string formulas. Another closely related work is Sparrow [21]. Similar to Janus, Sparrow uses over and under-approximation to produce equisatisfiable mutant formulas by the insertion of randomly synthesized subformulas. Sparrow also tested the SMT solvers Z3 and cvc5 and reportedly found around 80 bugs in various non-default configuration combinations and solver modes. The main conceptual difference is that mutants produced by Sparrow are less closely related than Janus's mutants since Sparrow replaces large terms within the seed formula. On the other hand, the idea behind Janus's ruleset is to make small incremental changes to the seed formula that are feasible for the developers to analyze. Unfortunately, Sparrow is not publicly available and rendering a thorough analysis of its technical differences from Janus impossible. Different from both approaches, Janus targets incompleteness bugs in SMT solvers in their default modes and only finds soundness and crash bugs as by-products.

Janus is also related to SMT solver performance fuzzing. Lascu et al. [2022] propose a metamorphic approach MF++ for fuzzing C++ libraries. It found 21 new bugs in the four SMT solver libraries (Z3, cvc5, Yices2, Boolector) and two Presburger arithmetic libraries (Omega & isl) including several incompleteness bugs. Incompleteness and performance bugs were also encountered in the context of program verifiers such as Dafny [109]. Mariposa [110] is a recent tool to quantify instability of SMT queries where instability is defined as a performance difference caused by assertion shuffling, symbol renaming, or under varying random seeds. Mariposa moreover features a benchmark set to evaluate proof stability. Besides domain-specific approaches to SMT and program verification, Janus is also related to general performance fuzzing. SlowFuzz [111] is an approach to finding complexity vulnerabilities in sorting and compression algorithms, PerfFuzz [112] is a tool using coverage guidance to find performance bugs along frequently executed program paths, and Ga-Proof [113] uses a genetic algorithm to detect performance bugs with inputs encoded as genes. Different from Janus, these approaches use runtime differences to identify bugs while Janus detects bugs based on the standard output of SMT solvers.

7

VALIDATING SMT SOLVERS FOR CORRECTNESS AND PERFORMANCE VIA GRAMMAR-BASED ENUMERATION

This chapter presents *Grammar-based Enumeration*, a systematic approach for validating and understanding SMT solvers. Realized by our tool ET, the approach has many complimentary benefits. Besides finding correctness bugs in SMT solvers, ET is specifically suitable for performance bugs. ET can be used to understand the evolution of solvers by deriving grammars for all major SMT theories. In a large-scale experiment, we test all releases of Z3 and CVC4/cvc5 from the last six years. The results suggest improved correctness in recent solver versions but decreased performance in newer releases of Z3 and regressions of early cvc5 releases.

7.1 MOTIVATION

Satisfiability modulo theory (SMT) solvers are foundational for many applications and systems in academia [8, 9, 10, 11, 12] and industry [16, 17, 52]. Hence, SMT solvers must be both *correct* and *performant*, particularly in safety-critical and security-critical domains. In the last several years, there has been much effort on improving SMT solvers, especially through fuzzing [1, 2, 21, 32]. Z3 and cvc5 are the two most powerful SMT solvers and are very reliable. Developers of Z3 and cvc5 fixed hundreds of correctness and performance bugs found by fuzzers. As a result of these and other fixes, SMT solvers have greatly matured. However, despite this, all existing fuzzers are unsystematic focusing on random testing. Unsystematic testing can lead to missed bugs and does not provide any guarantees. Consider, *e.g.*, the formula in Figure 7.1 which manifests a critical soundness bug in cvc5. The "declare-fun" statements specify two real variables, the "assert" specifies the constraints, and the "check-sat" queries the solver. The formula is satisfiable because for $a = -1$ the expression evaluates to $tan(sin(sin(-1))) \approx -0.923 > -1$. However, cvc5 incorrectly returns `unsat`. Apart from the soundness issue, it also uncovers a bug in the type-checker. Despite the simplicity of the bug, no ongoing fuzzing campaign, unit test, or user detected it. We reported this bug to cvc5's issue tracker. It got a "major" label and was promptly fixed by a cvc5 developer.

```
(declare-const a Int)
(assert (> (tan (sin (sin a))) a))
(check-sat)
```

Figure 7.1: Critical soundness bug in cvc5 found by our tool ET.
https://github.com/cvc5/cvc5/issues/10534

validating smt solvers via grammar-based enumeration This work changes the perspective on testing SMT solvers, advocating for systematic, grammar-based enumeration rather than random-based testing. We propose ET, a grammar-based enumeration tool for SMT solvers. We compile context-free grammars of the SMT theories to algebraic datatypes and leverage FEAT [33], an approach for functional enumeration. This realizes a test generator which ET couples with an oracle to perform differential testing between the solvers. Given a context-free grammar G, a number of tests N, and two or more SMT solvers, ET stress-tests each solver with the *N-smallest* inputs *w.r.t.* G. This approach has multiple unique benefits: (1) it exploits the *small scope hypothesis* which states that most bugs trigger on small-sized inputs [34, 35], (2) because of the small-sized bug triggers it is particularly suitable for identifying performance issues, and (3) it provides bounded correctness guarantees.[1] Our evaluation shows that ET is highly effective at bug finding in SMT solvers. We further demonstrate how ET can be used to understand the evolution of solvers, and thanks to its systematic nature and efficiency, we advocate its use for continuous integration.

bug hunting campaign Using ET, we conducted a large-scale fuzzing campaign for correctness and performance bugs in the state-of-the-art SMT solvers Z3 and cvc5. We reported 102 bugs among which 76 bugs were confirmed and 32 bugs were already fixed. We found bugs in various SMT theories, including arrays, floating points, real and integer arithmetic, strings *etc.* Even though SMT solvers have been extensively and continuously tested, we are still able to quickly find these bugs while the benefits of the other fuzzers seem to have saturated. We validated the developer's fixes including soundness, invalid models, crashes, and performance bugs. Among ET's soundness bug findings was another critical bug in cvc5-0.0.5's real arithmetic. The bug goes back to the major change from cvc4-1.8 to cvc5 and remained undetected for more than one and a half years. It was labeled "major" and was promptly fixed. Besides uncovering critical soundness

[1] ET provides bounded guarantees *w.r.t.*the grammar G and the differential oracle.

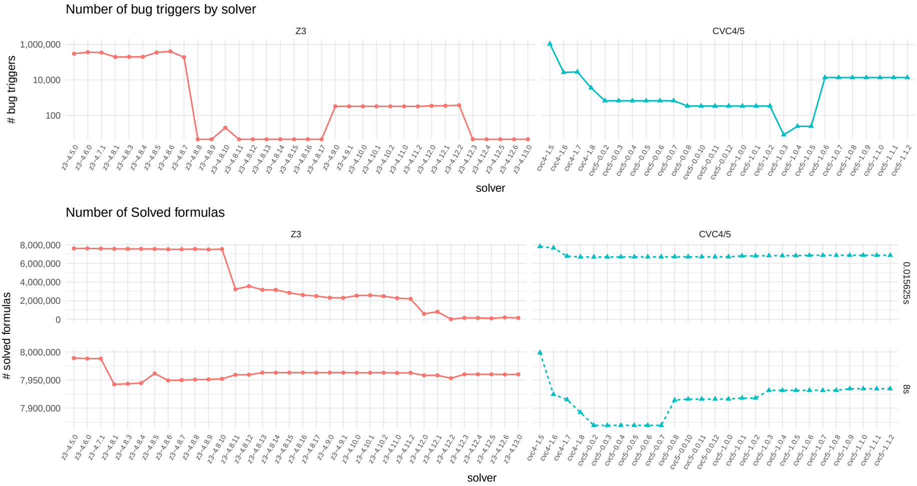


FIGURE 7.2: Evolution results for Z3 & CVC4/cvc5 releases from the last six years. Top: correctness in number of bug triggers. Bottom: performance in number of solved formulas.

bugs, a key advantage of ET's small-sized formulas is their suitability for identifying performance issues.

UNDERSTANDING THE EVOLUTION OF SMT SOLVERS Quantifying solver evolution helps developers understand long-term effects and users to judge particular features. With ET, we tested all consecutive versions of the SMT solvers Z3 and CVC4/cvc5 from the last six years (61 solvers). We devised eight grammars for the official SMT theories, generated one million formulas per grammar, and forwarded the formulas to the solvers. We tracked the solver's results and running time. Our correctness results reveal that both solvers have greatly matured (see Figure 7.2 top) with downward trends in the number of bug triggers. Perhaps most notably, both solvers have greatly matured in the theory of Strings, manifesting no bug triggers since many releases. This is striking as the theory of Strings was long considered to be among the most unstable.

For performance, we tracked the number of solved formulas from the lowest timeout of 0.015625s to the highest timeout of 8s. Lower timeouts help understand small aggregating effects while higher timeouts help understand performance regressions. For the lowest timeout 0.015625s, CVC4/cvc5's performance is roughly constant, but the performance of Z3 versions from 4.8.11 onwards worsened bottom. For the highest timeout of 8s, Z3 is roughly constant while cvc5's performance declines and then recovers. There is a decline from cvc4-1.8 to cvc5-0.0.2 caused by formulas in the Bitvector which is recovered in cvc5-0.0.8. Most recently, we observed regressions in the theory of Arrays beginning at cvc5-1.0.2 to cvc5-1.1.2.

PRACTICALITY OF ET AS A MONITORING TOOL We explore the practicality of ET for correctness and performance monitoring on commodity hardware. Investigating our data, we observe that 99% of bugs trigger within the first 120,000 formulas, and 80% occur within the first 51,000 formulas. We further observe that 40% of the total time is spent on the floating point theory. Exploiting these empirical facts, we can construct a pipeline that limits the formula count to 51,000 (120,000) and excludes the FP theory. Feasible realizations take three hours and 23 minutes for Z3 to cover 80% of the bugs, and one hour and 18 minutes for cvc5 to cover 99%.

7.2 ILLUSTRATIVE EXAMPLE

This section illustrates our approach. To utilize the functional enumeration capability of FEAT, a necessary step is to compile context-free grammars of the SMT theories to regular tree grammars. This realizes a grammar-based enumerator which we couple with a differential oracle for cross-checking the results of the SMT solvers under test. The following steps illustrate this.

1. DEVISE GRAMMAR FOR SMT THEORY. We first devise a context-free grammar for a dedicated SMT theory such as the theory of Arrays, which is important for many applications. We derive G_{Arrays} from a generic SMT-LIB grammar and include one array variable, one bitvector variable, and two constants (see Figure 7.3a).

2. COMPILE CONTEXT-FREE TO REGULAR TREE GRAMMAR. We compile the context-free grammar to a regular tree grammar as follows: for each production $lhs \rightarrow rhs_i$ in G_{Arrays}, we create a production $lhs \rightarrow C_i^{lhs} rhs_i$ with a fresh constructor C_i^{lhs}. E.g, consider the productions of nonterminal `arr_term` (see Figure 3b) which is compiled into an algebraic datatype of a functional programming language.

```
grammar Arrays;

type_arr:’(Array (_ BitVec 64) (_ BitVec 64))’;
type_bv: ’(_ BitVec 64)’;

bv_const: ’#x0000000000000000’
  | ’#x1111111111111111’;
var_a: ’a’;
var_b: ’b’;

uop_bv: ’bvnot’|’bvneg’;
bop_bv: ’bvand’|’bvor’|’bvadd’|’bvmul’
  |’bvudiv’|’bvurem’|’bvshl’|’bvlshr’;
bv_term: bv_const | var_b
  |’(’ uop_bv bv_term ’)’
  |’(’ bop_bv bv_term bv_term ’)’
  |’(select’ arr_term bv_term ’)’;

arr_term: var_a
 |’(store’ var_a bv_term bv_term ’)’;

bop_arr: ’=’ | ’distinct’;
bool_term: ’(’ bop_arr arr_term arr_term ’)’;

decl_csta:’(declare-const’ var_a type_arr ’)’;
decl_cstb:’(declare-const’ var_b type_bv ’)’;
assert_stmt:’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;

start: decl_csta decl_cstb assert_stmt check_sat;
```

(a) Grammar for the theory of Arrays G_{Arrays}

```
arr_term: var_a
 |’(store’ var_a bv_term bv_term ’)’;
```

↓

```
data Arr_term = C0_arr_term Var_a
| C1_arr_term P0 Store Var_a Bv_term Bv_term PC
```

(b) Compilation of a single production

```
(declare-const a (Array (_ BitVec 64) (_ BitVec 64)))
(declare-const b (_ BitVec 64))
(assert (= (store a (bvadd b b)
(bvadd b #x1111111111111111))(store (store a
#x1111111111111111 b) (bvadd b b) (bvneg b))))
(check-sat)
```

(c) φ_{Arrays} enumerated by ET (cvc5#8274)

cvc5-0.0.2	cvc5-0.0.3	cvc5-0.0.4	cvc5-0.0.5	cvc5-0.0.6	cvc5-0.0.7	cvc5-0.0.8	cvc5-0.0.10	cvc5-0.0.11	cvc5-0.0.12	cvc5-1.0.0	cvc5-1.0.1	cvc5-1.0.2	cvc5-1.0.3-9	cvc5-1.1.0-2
sat	sat	sat	sat	sat	sat	sat	sat	sat	sat	sat	sat	sat	unsat	unsat
✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓

(d) Solver results on formula φ_{Arrays}

FIGURE 7.3: Grammar-based enumeration with ET illustrated.

3. INTEGRATE REGULAR TREE GRAMMAR WITH FEAT AND ORACLE. We couple the regular tree grammar with the functional enumeration library FEAT. Given a desired number of tests N (*e.g.*, one million), leveraging FEAT, we enumerate the N-smallest formulas from G_{Arrays}.

4. RUNNING ET. We run ET with an oracle to cross-check all tests with the SMT solver Z3 and cvc5 writing correctness and performance bugs such as φ_{Arrays} to disk (Figure 7.3c).

The formula φ_{Arrays} is a real case. Z3 and cvc5 give different results on φ_{Arrays}. Z3 correctly returns `unsat` while cvc5 incorrectly reports `sat` on it. The bug has propagated seven releases and was fixed in cvc5 (Figure 7.3d). We have reported it to cvc5's issue tracker and it was fixed recently.

7.3 GRAMMAR-BASED ENUMERATION

This section (1) gives basic definitions, (2) formally introduces grammar-based enumeration, (3) describes ET's implementation, and (4) describes how we derive SMT theory grammars.

DEFINITIONS A *context-free grammar (CFG)* $G = \langle N, \Sigma, P, S \rangle$ consists of nonterminals N, terminals Σ, productions P, and a start symbol S from N. We assume G's productions to partition into two sets: productions yielding solely terminals P_Σ, and nonterminals P_N, respectively. A *regular tree grammar (RTG)* $G_{RTG} = \langle N', \Sigma', P', S' \rangle$ consists of nonterminals N', ranked alphabet Σ', productions P' and a start symbol S'. The elements in Σ' have constructors with arities. Terminals are realized as nullary constructors and constructors $C(\cdot)$ of strictly positive arity represent tree patterns. We view algebraic datatypes as instantiations of RTGs. The next definition shows how we compile a context-free grammar into the language of RTG.

Definition 7.3.1 (Context-free to regular tree grammar)**.** We compile context-free grammar $G = \langle N, \Sigma, P_\Sigma \cup P_N, S \rangle$ into regular tree grammar $RTG(G) = \langle N', \Sigma', P', S' \rangle$ as follows:

- $N' = N$ and $S' = S$
- $\Sigma' = \Sigma \cup \{ \ C_i^{lhs} | \quad p_i^{lhs} \in P_N \}$
- $P' = \{lhs \rightarrow C_i^{lhs} rhs_i \mid p_i^{lhs} \in P_N\} \cup P_\Sigma$

where $p_{lhs}^i \in P_N$ is the i-th production rule of the form $lhs \rightarrow rhs_i$.

The compilation enables to leverage FEAT, the powerful functional enumeration tool realizing close to random access to words in regular tree grammars. The following paragraph gives background on FEAT.

FUNCTIONAL ENUMERATION OF ALGEBRAIC TYPES [33] FEAT is a sized-based enumeration tool for algebraic datatypes. In a nutshell, FEAT's core functionality can be viewed as a function $FEAT : \mathbb{N} \rightarrow L(G_{RTG})$ from the natural numbers to the language of G_{RTG}. Each element $\varphi_i = FEAT(i)$ has an associated $size(\varphi_i)$ which is the number of nonterminals necessary to generate φ_i. E.g., consider the following formula of size 4 from the Core theory (see Figure 7.4a).

```
(declare-const a Bool) (declare-const b Bool) (assert false) (check-sat)
```

Algorithm 8 Pseudocode of ET's main prcoess.

```
1: procedure ET($G$, $\mathcal{O}$, $N$)                ▷ CFG $G$, oracle $\mathcal{O}$, number of tests $N$
2:     $bug\_triggers \leftarrow []$
3:     $G_{RTG} \leftarrow RTG(G)$                       ▷ Compilation based on Definition 7.3.1
4:     for $\varphi_i \in FEAT(G_{RTG})[0:N]$ do          ▷ Loop in parallel
5:         $bug\_found,\ trace_i \leftarrow \mathcal{O}(\varphi_i)$     ▷ Oracle check
6:         if $bug\_found$ then
7:             $bug\_triggers \leftarrow bug\_triggers.append((\varphi_i, trace_i))$
```

Four nonterminals are necessary to realize this formula. Hence changing `false` to `true`, or `a` to `b` will not increase size. However, recursion increases size by one *i.e.*, the following formula has a size of five:

```
(declare-const a Bool) (declare-const b Bool) (assert (not false)) (check-sat)
```

FEAT realizes a partition of $L(G_{RTG})$ based on size. Moreover, FEAT realizes random access to elements in $L(G_{RTG})$. We write $FEAT(G_{RTG})[0 : N]$ to denote the N-smallest elements of G_{RTG}.

ET's realization We next describe ET's realization. ET takes as inputs a context-free grammar G, an oracle $\mathcal{O}$, and a desired number of tests N. The following pseudocode shows ET's main process (Figure 8). ET starts by initializing a list *bug_triggers* (Line 2). Next, it compiles the context-free grammar G to regular tree grammar G_{RTG} (Line 3). Then, we use FEAT to generate N tests through which we iterate (Line 4). For every formula φ_i, we perform an oracle check calling the SMT solvers. If the oracle detects a bug, we save the bug trigger φ_i along with a trace $trace_i$ (Line 7). The oracle can be implemented in different ways, *i.e.*, by differentially testing, calling a certified solver *etc.* ET is implemented in 103 lines of Python and Bash.

deriving grammars for the smt theories A key ingredient of our approach are the grammars. We derive one grammar per official SMT theory including the booleans (Core), integers (Ints), real numbers (Reals), mixed reals and integers (RealInts), bitvectors (FixedSizeBitVectors), arrays (ArraysEx), floating point numbers (FP), and unicode strings (Strings). As a template, we use a generic SMT-LIB grammar from the ANTLR grammar repository. [2] We studied the theory specifications to ensure that

[2] https://github.com/antlr/grammars-v4/blob/master/smtlibv2/SMTLIBv2.g4

```
grammar Core;
type_bool: ’Bool’;
bool_const: ’true’|’false’;
var: ’a’|’b’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
bool_term: bool_const | var
  |’(not’ bool_term ’)’
  |’(’ binop_bool bool_term bool_term ’)’
  |’(ite’ bool_term bool_term bool_term ’)’;
decl_csts: ’(declare-const’ var type_bool ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csts assert_stmt check_sat;
```

(a) Core

```
grammar Arrays;
type_arr: ’(Array (_ BitVec 64) (_ BitVec 64))’;
type_bv: ’(_ BitVec 64)’;
bv_const: ’#x0000000000000000’
  |’#x1111111111111111’;
var_a: ’a’;
var_b: ’b’;
uop_bv: ’bvnot’|’bvneg’;
bop_bv: ’bvand’|’bvor’|’bvadd’|’bvmul’
  |’bvudiv’|’bvurem’|’bvshl’|’bvlshr’;
bv_term: bv_const | var_b
  |’(’ uop_bv bv_term ’)’
  |’(’ bop_bv bv_term bv_term ’)’
  |’(select’ arr_term bv_term ’)’;
arr_term: var_a
  |’(store’ var_a bv_term bv_term ’)’;
bop_arr: ’=’|’distinct’;
bool_term: ’(’ bop_arr arr_term arr_term ’)’;
decl_csta: ’(declare-const’ var_a type_arr ’)’;
decl_cstb: ’(declare-const’ var_b type_bv ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csta decl_cstb assert_stmt check_sat;
```

(b) Arrays

```
grammar Bitvectors;
type_bv: ’(_ BitVec 64)’;
bv_const: ’#x0000000000000000’
  | ’#x1111111111111111’;
var: ’a’ | ’b’;
uop_bv: ’bvnot’|’bvneg’;
bop_bv: ’bvand’|’bvor’|’bvadd’|’bvmul’|’bvudiv’
  |’bvurem’|’bvshl’| ’bvlshr’;
bv_term : bv_const | var
  | ’(’ uop_bv bv_term ’)’
  | ’(’ bop_bv bv_term bv_term ’)’;
int_const: ’0’ | ’1’;
int_term : int_const | ’(bv2nat’ bv_term ’)’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
binop_real_bool: ’=’|’>’|’<’|’>=’|’<=’;
binop_bv_bool: ’bvult’| ’=’|’distinct’;
binop_bv_int: ’=’|’distinct’;
bool_term: ’(not’ bool_term ’)’
  | ’(’ binop_bool bool_term bool_term ’)’
  | ’(ite’ bool_term bool_term bool_term ’)’
  | ’(’ binop_bv_bool bv_term bv_term ’)’
  | ’(’ binop_bv_int int_term int_term ’)’;
decl_csts: ’(declare-const’ var type_bv ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csts assert_stmt check_sat;
```

(c) Bitvectors

```
grammar Ints;
type_int: ’Int’;
int_const: ’0’|’1’;
var: ’a’|’b’;
uop_int: ’-’|’abs’;
bop_int: ’-’|’+’|’*’|’div’|’mod’;
int_term: int_const | var
  | ’(’ uop_int int_term ’)’
  | ’(’ bop_int int_term int_term ’)’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
binop_int_bool: ’=’|’>’|’<’|’>=’|’<=’;
bool_term: ’(not’ bool_term ’)’
  | ’(’ binop_bool bool_term bool_term ’)’
  | ’(ite’ bool_term bool_term bool_term ’)’
  | ’(’ binop_int_bool int_term int_term ’)’;
decl_csts: ’(declare-const’ var type_int ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csts assert_stmt check_sat;
```

(d) Ints

```
grammar Reals;
type_real: ’Real’;
real_const: ’0.0’|’1.0’;
var: ’a’|’b’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
binop_real_bool: ’=’|’>’|’<’|’>=’|’<=’;
uop_real: ’sin’|’cos’|’tan’;
binop_real: ’-’| ’+’|’*’|’/’| ’mod’;
real_term: real_const | var
  | ’(’ uop_real real_term ’)’
  | ’(’ binop_real real_term real_term ’)’;
bool_term: ’(not’ bool_term ’)’
  | ’(’ binop_bool bool_term bool_term ’)’
  | ’(ite’ bool_term bool_term bool_term ’)’
  | ’(’ binop_real_bool real_term real_term ’)’;
decl_csts: ’(declare-const’ var type_real ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_cst assert_stmt check_sat;
```

(e) Reals

```
grammar FP;
type_fp: ’(_ FloatingPoint 11 53)’;
fp_const: ’(fp #b0 #b0{11} #b0{64})’
  | ’(fp #b1 #b1{11} #b1{64})’;
var: ’a’|’b’;
rm: ’RNE’|’RNA’|’RTP’|’RTN’|’RTZ’;
bop_bool: ’=’|’distinct’|’fp.leq’|’fp.lt’|’fp.eq’
  |’fp.geq’|’fp.leq’|’fp.gt’|’fp.lt’;
uop_fp: ’fp.abs’|’fp.neg’;
bop_fp: ’fp.rem’|’fp.min’|’fp.max’;
top_rm_fp: ’fp.add’|’fp.sub’|’fp.mul’
  |’fp.div’|’fp.fma’;
bop_rm_fp: ’fp.sqrt’|’fp.roundToIntegral’;
fp_term: fp_const | var
  | ’(’ uop_fp fp_term ’)’
  | ’(’ top_rm_fp rm fp_term fp_term ’)’
  | ’(’ bop_rm_fp rm fp_term ’)’
  | ’(’ bop_rm_fp rm fp_term ’)’
  | ’(’ bop_fp fp_term fp_term ’)’;
bool_term: ’(’ bop_bool fp_term fp_term ’)’;
decl_csts: ’(declare-const’ var type_fp ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csts assert_stmt check_sat;
```

(f) FP

Figure 7.4: Derived grammars for the SMT theories.

```
grammar RealInts;
type_real: ’Real’;
type_int: ’Int’;
int_const: ’0’|’1’;
real_const: ’0.0’|’1.0’;
var_a: ’a’;
var_b: ’b’;
uop_int: ’-’|’abs’;
bop_int: ’-’|’+’|’*’|’div’|’mod’;
uop_real_int: ’to_int’;
int_term: int_const | var_a
  | ’(’ uop_int int_term ’)’
  | ’(’ binop_real int_term int_term ’)’
  | ’(’ uop_real_int ’)’;
uop_real: ’sin’ |’cos’ |’tan’;
binop_real: ’-’ | ’+’ | ’*’ | ’/’ | ’mod’;
real_term: real_const | var_b
  | ’(’ uop_real real_term ’)’
  | ’(’ binop_real real_term real_term ’)’
  | ’(’ ’to_real’ real_term ’)’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
binop_real_bool: ’=’ | ’>’ | ’<’ | ’>=’ | ’<=’;
bool_term: ’(not’ bool_term ’)’
  | ’(’ binop_bool bool_term bool_term ’)’
  | ’(ite’ bool_term bool_term bool_term ’)’
  | ’(’ binop_real_bool real_term real_term ’)’;
decl_csta: ’(declare-const’ var_a type_int ’)’;
decl_cstb: ’(declare-const’ var_b type_real ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csta decl_cstb assert_stmt check_sat;
```

(a) RealInts

```
grammar Strings;
type_str: ’String’;
str_const: ’""’ | ’"a"’;
int_const: ’0’|’1’;
regex_const: ’re.none’|’re.all’|’re.allchar’;
var: ’a’ | ’b’;
bop_str: ’str.++’;
top_str: ’str.replace’ | ’str.replace_all’;
uop_int_str: ’str.from_int’;
uop_regex:’re.comp’|’re.+’|’re.opt’;
bop_regex:’re.union’|’re.inter’|’re.++’|’re.diff’;
uop_str_regex: ’str.to_re’| ’re.range’;
binop_bool: ’and’|’or’|’xor’|’=’|’distinct’;
bop_str_bool: ’=’|’distinct’|’str.<=’
  |’str.prefixof’|’str.suffixof’|’str.contains’;
str_term: str_const | var
  | ’(’ top_str str_term str_term str_term ’)’
  | ’(str.at’  str_term int_term ’)’
  | ’(str.substr’ str_term int_term int_term ’)’
  | ’(’ uop_int_str int_term ’)’;
int_term: int_const
  | ’(str_to_int’ str_term ’)’
  | ’(str.indexof’ str_term str_term int_term ’)’;
regex_term: regex_const
  | ’(’ uop_regex regex_term ’)’
  | ’(’ bop_regex regex_term regex_term ’)’
  | ’(’ uop_str_regex str_term ’)’
  | ’(re.*’ regex_const ’)’;
bool_term: ’(not’ bool_term ’)’
  | ’(’ binop_bool bool_term bool_term ’)’
  | ’(ite’ bool_term bool_term bool_term ’)’
  | ’(’ bop_str_bool str_term str_term ’)’
  | ’(’ ’str.is_digit’ str_term ’)’
  | ’(’ ’str.in_re’ str_term regex_term ’)’;
decl_csts: ’(declare-const’ var type_str ’)’;
assert_stmt: ’(assert’ bool_term ’)’;
check_sat: ’(check-sat)’;
start: decl_csts assert_stmt check_sat;
```

(b) Strings

FIGURE 7.5: Derived grammars for the SMT theories (ctd).

the grammars covered all operators from the respective theories. [3] We include two variables and two constants: *e.g.*,"`true`”,”`false`" for Core, "`0`", "`1`" for Ints, "`0.0`", "`1.0`" for Reals, "`#x0000000000000000`", "`#x1111111111111111`" for Bitvectors and Arrays, "`""`", "`"a"`" for Strings, and "`(fp #b0 #b0{11} #b0{64})`", "`(fp #b1 #b1{11} #b1{64})`" for FP. The grammars describe SMT-LIB scripts with a variable declaration block followed by a single assert and a check-sat command. We emphasize that our approach is not restricted to these grammars (see Figure 7.4 + 7.5). Richer grammars of SMT theories can be devised by modifying existing or creating new grammars.

7.4 EMPIRICAL EVALUATION

This section details our extensive evaluation with ET demonstrating the practical effectiveness of grammar-based enumeration for testing SMT solvers. We first evaluate ET through a bug-hunting campaign on the trunk versions of the state-of-the-art SMT solvers Z3 and cvc5. Using ET, we then investigate the evolution of all stable solvers releases over the last six years. We finally explore ET’s potential as a monitoring tool for continuous integration of solver commits.

RESULT SUMMARY

- *Many bugs in Z3 and cvc5*: We found 102 bugs, 53 correctness and 49 performance bugs. Among these 76 were confirmed, and 32 were fixed by the developers.

- *Insightful evolution results:* We observe significantly increased correctness of Z3 and cvc5 within the last six years; For performance, recent Z3 releases have regressed on short timeouts, while early cvc5 releases regressed on long timeouts.

- *Practicality for continuous integration:* ET is practical for continuous integration: it covers 99% of the cvc5 bugs (found in RQ2) in less than two hours, and 80% of the Z3 bugs in less than four hours on a commodity CI/CD pipeline.

Research Questions

We aim to answer the following four consecutive research questions:

[3] https://smtlib.cs.uiowa.edu/theories.shtml

RQ1 *How effective is ET at bug finding?*

RQ2 *Can we use ET to quantify the correctness of SMT solvers?*

RQ3 *Can we use ET to quantify the performance of SMT solvers?*

RQ4 *How practical is ET for continous integration?*

RQ2 and RQ3 are motivated by ET's systematic testing and the small scope hypothesis which states that most interesting behavior of software is observable on small inputs. Quantifying solver evolution lets developers observe long-term effects and helps users in making better choices for their apps *i.e.* choosing a solver for a particular theory, judging the state of a solver feature *etc.*

Evaluation Setup

For all experiments, we used a machine equipped with an AMD EPYC 9654 CPU with 96 cores and 64GB RAM running an Ubuntu 22.04.4 LTS (64-bit). We disabled simultaneous multi-threading and frequency scaling for more consistent performance. For RQ2-RQ4, we repeated the experiments two times and averaged the results.

ORACLES We use the following two oracles for our evaluation with ET:

$\mathcal{O}_{\text{TEST}}$ is a differential oracle with daily-builds of the SMT solvers Z3 and cvc5. The oracle calls the solvers in the following order: Z3 in default mode, cvc5 in default modes, Z3's new core, Z3 with further options, cvc5 with further options, and cvc4-1.8 for catching longstanding regressions in cvc5. The first terminating solver call serves as the reference to all others. We use a timeout of 60s on all solver calls.

$\mathcal{O}_{\text{EVOL}}$ is a differential oracle with all SMT solvers Z3 and CVC4/cvc5 releases from November 2016 to March 2024 making 61 solvers in total. The oracle calls all solvers and uses the latest cvc5 as the reference for Z3 releases and the latest Z3 as a reference for all cvc5 and CVC4 releases. We use a timeout of 8s on all solver calls.

All configurations in both oracles were run with model validation and unsat cores checks. Oracle $\mathcal{O}_{\text{test}}$ is used in RQ1 and oracle $\mathcal{O}_{\text{evol}}$ is used in RQ2-RQ4. For the fuzzing campaign in RQ1, we extended the basic grammars to up to five variables and two asserts.

Status	Z3	cvc5	Total
Reported	38	64	102
Confirmed	15	61	76
Fixed	13	19	32
Duplicate	0	2	2
Won't fix	4	1	5

(a)

Type	Z3	cvc5	Total
Soundness	2	10	12
Invalid Model	7	10	17
Crash	1	19	20
Performance	5	22	27

(b)

#Options	Z3	cvc5	Total
default	7	34	41
1	6	27	33
2	2	0	2

(c)

Figure 7.6: (a) Status of bugs found in Z3 and cvc5 with ET, (b) bug types among the confirmed bug, (c) number of options supplied to Z3 and cvc5 among the confirmed bugs.

RQ1: How effective is ET at bug finding?

Using ET with oracle $\mathcal{O}_{\text{test}}$, we extensively stress-tested the SMT solvers Z3 and cvc5. We reported 102 bugs, out of which 76 were confirmed, 32 were fixed (see Figure 7.6a). The bug types are fairly evenly distributed: 12 are soundness bugs, 17 are invalid models, 20 are crashes, and 27 are performance bugs (see Figure 7.6b). Of the confirmed bugs, most bugs affect the solver's default modes (41 out of 76), followed by single-option configurations (33 out of 76) and two bugs affect two option configurations (see Figure 7.6c).[4] We also inspect the theory distribution which we analyze for correctness bugs and performance bugs separately. Among the bugs that we reported, there are 53 correctness and 49 performance bugs. Among the correctness bugs, we observe most bugs in Reals (17 out of 53) followed by Arrays (13 out of 53), followed by FP (10 out of 53), and Ints (6 out of 53). Breaking it down further by solver, we observe that Z3 has most correctness bugs in FP (5) followed by Reals (4) while cvc5 has most bugs in Reals (13) and Arrays (11). Among the performance bugs, most bugs occur in Strings (20 out of 49) followed by Ints (14 out of 49), and Bitvectors (8 out of 49). Breaking it down further by solver, most performance bugs in Z3 occur in Strings (10), followed by Ints (9) while most performance bugs in cvc5 occur in Strings (10) and Reals (8). Despite being careful while reporting bugs, there were also 2 duplicates and 5 won't fix reports. The duplicates were two bugs that were earlier findings by cvc5's internal fuzzer Murxla [30], the won't fixes consist of four bugs in Z3's new core considered "too early" and an inconsistency of cvc5 and cvc4 with cvc4 being unsound. As an intermediate conclusion, we observe that ET found almost twice as

[4] Both two-option bugs are related to Z3's new core `tactic.default_tactic=smt sat.euf=true`.

many bugs in cvc5 as compared to Z3. A partial explanation for this could be the major overhauls from cvc4 to cvc5, extending previous reports of performance regressions in cvc5 [53] to correctness. We moreover observe that ET found most bugs in the default modes of the solvers demonstrating ET's effectiveness. Strikingly none of the concurrent fuzzing campaigns, unit tests, or users have found the simple bugs that ET found. It is remarkable that ET can find so many bugs given the extensive and continuous testing of other fuzzers. To showcase the simplicity and diversity of ET's findings we detail multiple bug samples from our bug-hunting campaign with ET.

SOUNDNESS BUG IN DEFAULT CVC5 (FIGURE 7.7A) The formula realizes a conjunction of two equations. The first equation `(= a 0)` is satisfied when variable `a` is zero. The second equation `(= b (cos a))` is satisfied if `b` equals the result of `(cos a)`, which in turn has to be equal 1 to satisfy the first equation. Setting `a = 0` and `b = 1` satisfies the whole formula. However, cvc5 returns `unsat` on this formula, which is incorrect. The developers promptly inspected and fixed this bug. The associated pull request was labeled with "major" underpinning its criticality. The bug was undiscovered for two and a half years propagating from cvc4-1.8 to cvc5-0.0.7.

SOUNDNESS BUG IN Z3'S NEW CORE (FIGURE 7.7B) The formula realizes the inequality `-a > (1 mod -1) = 0`. Clearly, a negative `a` would satisfy the inequation, hence the formula is satisfiable. However, Z3's new core reports `unsat` for this formula.

SOUNDNESS BUG IN CVC5'S STRING THEORY (FIGURE 7.7C) The formula triggers a soundness bug in cvc5's string theory. cvc5 with option `--strings-eager-len-re` incorrectly returns `unsat` on this formula, although it is satisfiable. The pull request fixing this bug got a "major" label.

SOUNDNESS BUG IN Z3 (FIGURE 7.7D) The formula triggers a soundness bug in Z3's array theory. Z3 with disabled bitvector equality axioms, incorrectly returns `sat` on the formula, while cvc5 gives `unsat`, the correct result. The issue was 1.5-year latent; it has existed since Z3 version 4.8.9. We reported the issue and it was promptly fixed by Z3's main developer. The bug trigger is sizable; almost all other bugs had a smaller size.

PERFORMANCE BUG IN Z3'S INTS THEORY (FIGURE 7.7E). The formula triggers a performance bug in Z3. The formula has a single integer variable

```
(declare-fun a () Real)
(declare-fun b () Real)
(assert (and (= a 0) (= b (cos a))))
(check-sat)
```

(a) Soundness bug in default cvc5: bug in non-linear real-arithmetic.

https://github.com/cvc5/cvc5/issues/7948

```
(declare-const a Int)
(assert (> (- a) (mod 1 (- 1))))
(check-sat)
```

(b) Soundness bug in z3's new core: bug in non-linear real-arithmetic.

https://github.com/Z3Prover/z3/issues/6116

```
(declare-const a String)
(assert (str.in_re a (re.++ (re.opt
re.allchar) (re.diff (re.* re.none)
(str.to_re a)))))
(check-sat)
```

(c) Soundness bug in cvc5: Issue in eager string solving component.

https://github.com/cvc5/cvc5/issues/8548

```
(declare-const a
(Array (_ BitVec 64) (_ BitVec 64)))
(declare-const b (_ BitVec 64))
(assert (= (store (store a b b)
(select a b)(select a b)) (store
(store a b #x1111111111111111)
#x1111111111111111
(bvudiv b #x1111111111111111))))
(check-sat)
```

(d) Soundness Bug in Z3: Z3 returns `sat` on this unsatisfiable formula.

https://github.com/Z3Prover/z3/issues/2391

```
(declare-const a Int)
(assert (not (is_int (- (* a a )))))
(check-sat)
```

(e) Performance bug in default Z3: timeout on a simple unsatisfiable formula.

https://github.com/Z3Prover/z3/issues/6800

```
(declare-const a Real)
(assert (>= (- a) (cos 1.0)))
(check-sat)
```

(f) Performance bug in default cvc5: timeout on simple real formula.

https://github.com/cvc5/cvc5/issues/9873

```
(declare-const a String)
(assert (str.contains
(str.replace_all a "a" "") "a"))
(check-sat)
```

(g) Performance bug in cvc5: timeouts on simple string formula.

https://github.com/cvc5/cvc5/issues/9875

```
(declare-const a (_ BitVec 64))
(assert (= a (bvurem (bvnot a) a)))
(check-sat)
```

(h) Performance bug in Z3 and cvc5: timeouts on bitvector formula.

https://github.com/Z3Prover/z3/issues/6800
https://github.com/cvc5/cvc5/issues/9874

FIGURE 7.7: Selected correctness and performance bugs found by ET.

`a`, and the function `is_int` checks whether its argument is an integer or not.

Since this expression is integral, the formula should be `unsat`. However, the z3 trunk version times out on this formula.

PERFORMANCE BUG IN CVC5'S REAL THEORY (FIGURE 7.7F) The formula triggers a performance bug in cvc5's Real theory. Although it is `sat`, i.e. any integer greater than the negative of `cos(1)` would solve it. Despite this, cvc5 times out on the formula.

PERFORMANCE BUG IN CVC5 STRING (FIGURE 7.7G) The simple string formula is clearly unsatisfiable, as variable `a` cannot contain the string "a" if all occurrences of "a" are replaced by the empty string using `str.replace_all`. However, cvc5 times out on this formula. We reported the bug and it was confirmed by a cvc5 developer.

PERFORMANCE BUG IN BOTH Z3 AND CVC5 (FIGURE 7.7H) The formula triggers a performance bug in both Z3 and cvc5. It is a simple bitvector expression, on which both solvers time out. The issue was confirmed by cvc5 and is still open in Z3.

Result #1: *ET is highly effective at bug finding: we found 102 bugs in the trunk versions of Z3 and cvc5 with most bugs in the default modes of the solvers. Notably, ET found these bugs despite the extensive and continuous testing.*

RQ2: Can we use ET to quantify the correctness of SMT solvers?

Having observed ET's effectiveness at bug finding, a natural follow-up question is whether ET's systematic testing can be used to quantify the correctness of SMT solvers. To approach this question, we ran ET with oracle $\mathcal{O}_{\text{evol}}$ on all consecutive releases of Z3 and CVC4/cvc5 from the last six years (see Figure 7.8). As a reference for validating the results of the solvers, we used the latest Z3 version (z3-4.13.0) for all CVC4/cvc5 solvers and the latest CVC4/cvc5 version (cvc5-1.1.2) as a reference for all Z3 versions. We then compare the number of bug triggers, *i.e.*, failing tests, per solver. For all solver calls, we chose a timeout of 8 seconds.

NUMBER OF BUG TRIGGERS PER SOLVER AND THEORY We present the results in a line plot (Figure 7.9) with two columns, one for each solver Z3 and CVC4/cvc5. The rows correspond to the different theories, e.g., Core, Ints, Reals, *etc.* All rows share the horizontal axis with SMT solver releases

Solver	date	solver	date	solver	date	solver	date
z3-4.5.0	Nov 2016	z3-4.8.11	Jul 2021	z3-4.8.17	May 2022	z3-4.12.2	May 2023
cvc4-1.5	Jul 2017	z3-4.8.12	Jul 2021	cvc5-0.0.7	May 2022	cvc5-1.0.6	Aug 2023
z3-4.6.0	Dec 2017	cvc5-0.0.2	Oct 2021	z3-4.9.1	Jun 2022	cvc5-1.0.7	Aug 2023
z3-4.7.1	May 2018	cvc5-0.0.3	Oct 2021	z3-4.9.0	Jun 2022	cvc5-1.0.8	Aug 2023
cvc4-1.6	Jun 2018	z3-4.8.13	Nov 2021	z3-4.10.0	Jun 2022	cvc5-1.0.9	Dec 2023
z3-4.8.1	Oct 2018	cvc5-0.0.4	Nov 2021	z3-4.10.1	Jun 2022	cvc5-1.1.0	Dec 2023
z3-4.8.3	Nov 2018	z3-4.8.14	Dec 2021	z3-4.10.2	Jun 2022	z3-4.12.3	Dec 2023
z3-4.8.4	Dec 2018	cvc5-0.0.5	Jan 2022	cvc5-1.0.1	Jul 2022	z3-4.12.4	Dec 2023
cvc4-1.7	Apr 2019	cvc5-0.0.6	Jan 2022	z3-4.11.0	Aug 2022	cvc5-1.1.1	Jan 2024
z3-4.8.5	Jun 2019	z3-4.8.15	Mar 2022	cvc5-1.0.2	Aug 2022	z3-4.12.5	Jan 2024
z3-4.8.6	Sep 2019	cvc5-0.0.8	Mar 2022	z3-4.11.2	Sep 2022	z3-4.12.6	Feb 2024
z3-4.8.7	Nov 2019	cvc5-0.0.10	Apr 2022	cvc5-1.0.3	Dec 2022	cvc5-1.1.2	Mar 2024
z3-4.8.8	May 2020	cvc5-0.0.11	Apr 2022	z3-4.12.0	Jan 2023	z3-4.13.0	Mar 2024
cvc4-1.8	Jun 2020	cvc5-0.0.12	Apr 2022	z3-4.12.1	Jan 2023		
z3-4.8.9	Sep 2020	cvc5-1.0.0	Apr 2022	cvc5-1.0.4	Jan 2023		
z3-4.8.10	Jan 2021	z3-4.8.16	Apr 2022	cvc5-1.0.5	Mar 2023		

Figure 7.8: Z3 and CVC4/cvc5 versions from November 2016 to March 2024 used in RQ2 and RQ3. In grey: solvers used for cross-checking in ET.

from the oldest to the newest (left to right). For each row, the vertical axis denotes the bug trigger counts per theory and solver in a logarithmic scale. Unsoundness bugs are depicted in red, invalid model bugs are depicted in green, and crash bugs are depicted in blue. Additionally, we present overview correctness results in a table on the next page (c.f. Figure 7.10).

Z3 Considering the correctness results of Z3 (Figure 7.9 left), we observe a striking decrease in bug triggers. In its oldest release 4.5.0, there were bugs in 5 out of 8 theories including critical soundness bugs in Strings, and FP. By contrast, in the most recent version 4.13.0, there are no bug triggers at all, most importantly, no soundness bugs. Examining further, we observe that Z3 became significantly more correct even in the theory of Strings. It is now reliable since many releases. This is remarkable as the theory of Strings was long considered unstable in both solvers. Another interesting finding is the sudden decrease in bug triggers from version 4.8.7 to 4.8.8. While in version 4.8.7, there are bug triggers in 5 out of 8 theories, in 4.8.8 there are no bugs. Besides 4.8.8 and 4.8.9, the other Z3 versions without bugs are z3-4.8.11 until z3-4.8.17 and then z3-4.12.3 until 4.13.0.

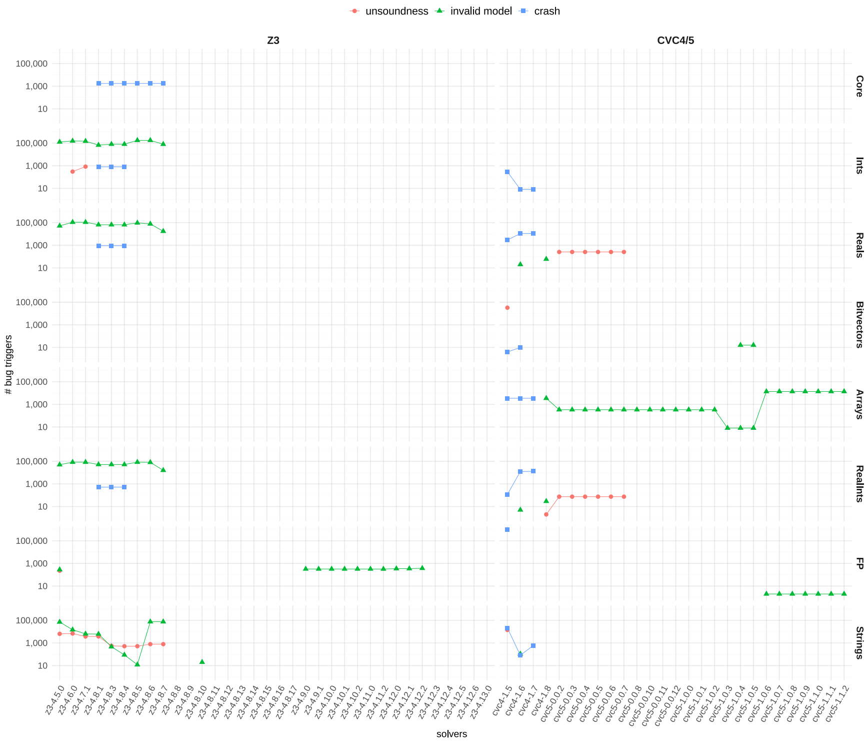


FIGURE 7.9: Bug triggers per theory in all releases of Z3 (left) and CVC4/cvc5 (right) since November 2016.

CVC4/CVC5 For the correctness results of CVC4/cvc5, we likewise see a striking decrease in bugs triggered by ET. Its oldest release (cvc4-1.5) has bugs in 7 out of 8 theories including critical soundness bugs in Strings and Bitvectors. On the other hand, the latest release (cvc5-1.1.2) only exhibits invalid model bugs in Arrays and FP. Notably, there are soundness bug triggers in Ints and RealInts beginning at cvc4-1.8 propagating to early versions of cvc5. Similar to Z3, we also observe that bug triggers in cvc5's string theory have significantly decreased.

Result #2: *Enabled by ET, we found that Z3 and CVC4/cvc5's correctness has significantly improved in (almost) all theories. Notably, the theory of Strings is now stable in both solvers since many releases.*

Solver	unsound	inv. model	crash
z3-4.13.0	0.0	0.0	0.0
z3-4.12.3-6	0.0	0.0	0.0
z3-4.12.2	0.0	370.5	0.0
z3-4.12.1	0.0	347.5	0.0
z3-4.12.0	0.0	346.0	0.0
z3-4.11.2	0.0	319.5	0.0
z3-4.11.0	0.0	320.5	0.0
z3-4.10.2	0.0	321.0	0.0
z3-4.10.1	0.0	321.0	0.0
z3-4.10.0	0.0	321.0	0.0
z3-4.9.1	0.0	321.5	0.0
z3-4.9.0	0.0	322.0	0.0
z3-4.8.11-17	0.0	0.0	0.0
z3-4.8.10	0.0	20.0	0.0
z3-4.8.9	0.0	0.0	0.0
z3-4.8.8	0.0	0.0	0.0
z3-4.8.7	785.0	187854.0	1759.0
z3-4.8.6	785.0	406639.5	1759.0
z3-4.8.5	516.0	352492.0	1742.0
z3-4.8.4	516.0	196600.0	3982.0
z3-4.8.3	554.0	196974.0	3982.0
z3-4.8.1	3750.0	189173.0	3982.0
z3-4.7.1	4556.0	345172.0	0.0
z3-4.6.0	7134.0	360773.5	0.0
z3-4.5.0	6619.0	296773.0	0.0

Solver	unsound	inv. model	crash
cvc5-1.1.2	0.0	13598.0	0.0
cvc5-1.1.1	0.0	13598.0	0.0
cvc5-1.1.0	0.0	13598.0	0.0
cvc5-1.0.9	0.0	13598.0	0.0
cvc5-1.0.8	0.0	13598.0	0.0
cvc5-1.0.7	0.0	13598.0	0.0
cvc5-1.0.6	0.0	13598.0	0.0
cvc5-1.0.5	0.0	24.0	0.0
cvc5-1.0.4	0.0	24.0	0.0
cvc5-1.0.3	0.0	8.0	0.0
cvc5-1.0.2	0.0	336.0	0.0
cvc5-1.0.1	0.0	336.0	0.0
cvc5-1.0.0	0.0	336.0	0.0
cvc5-0.0.12	0.0	336.0	0.0
cvc5-0.0.11	0.0	336.0	0.0
cvc5-0.0.10	0.0	336.0	0.0
cvc5-0.0.8	0.0	336.0	0.0
cvc5-0.0.7	330.0	336.0	0.0
cvc5-0.0.6	330.0	336.0	0.0
cvc5-0.0.5	330.0	336.0	0.0
cvc5-0.0.4	330.0	336.0	0.0
cvc5-0.0.3	330.0	336.0	0.0
cvc5-0.0.2	330.0	336.0	0.0
cvc4-1.8	2.0	3523.0	0.0
cvc4-1.7	0.0	0.0	28152.0
cvc4-1.6	0.0	133.0	26528.0
cvc4-1.5	46489.5	0.0	1027886.0

Figure 7.10: Bug triggers in Z3 (left) and CVC4/cvc5 releases (right). Shaded: solvers in which ET found no bugs.

RQ3: Can we use ET to quantify the performance of SMT solvers?

Having noticed the improved correctness of the solvers, we next investigate their performance. We first examine the number of solved formulas for short and long timeouts. As a second step, we then examine the runtime for jointly solved formulas and the throughput.

NUMBER OF SOLVED FORMULAS We evaluate the number of solved formulas for different timeouts ranging from the lowest (T=0.015625s) to the highest (T=8s) in powers of two. For the lowest timeout of T=0.015625s, we show a bar plot (see Figure 7.11). We have a column for each solver

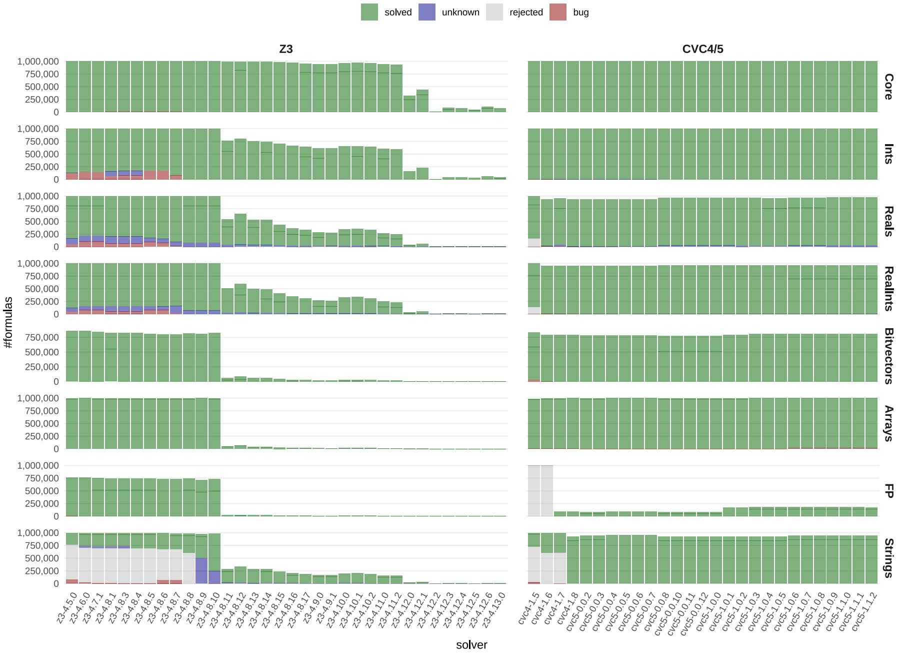


Figure 7.11: Number of formulas for lowest timeout T = 0.015625s. White spaces indicate unsolved formulas.

Z3 and CVC4/cvc5. The rows correspond to the different theories. All columns share the horizontal axis on which the SMT solver releases are listed from old to new (left to right). For each row, the vertical axis denotes solved formulas. For the highest timeout T = 8s, we show a line plot (see Figure 7.13). For a more complete set of plots, we refer to Appendix A.4.

LOWEST TIMEOUT T=0.015625S Considering the results for Z3 (Figure 7.11 left), we see a significant decrease from earlier to later releases in the number of solved formulas. This is especially true for Bitvectors, Arrays, Strings, and FP. To a lesser extent also for Ints, Reals, and RealInts. The most significant effect manifests from z3-4.8.10 to z3-4.8.11. Less significant decreases occur from 4.11.2 to 4.12.0 and 4.12.1 to 4.12.2 respectively. There is a significant increase in solved formulas for the theory of Strings from version z3-4.8.8 to z3-4.8.10. This is caused by a large set of formerly rejected formulas that were solved in z3-4.8.10. In z3-4.8.9, the version in between,

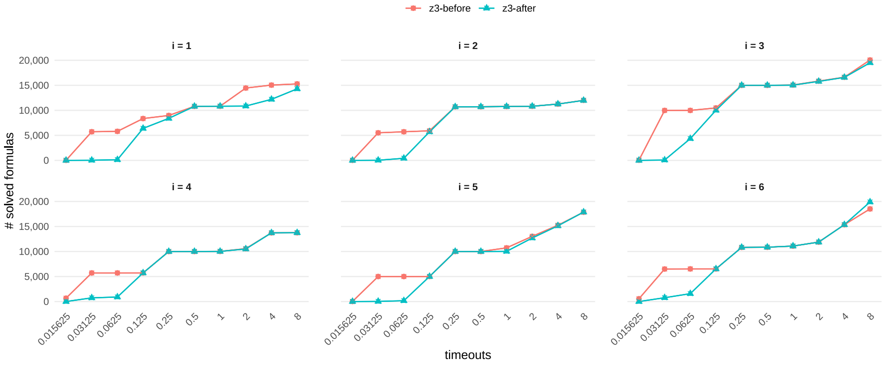


Figure 7.12: Results on larger formulas (4 KB and larger) to understand the significant performance decline in Z3: before (z3-before) and after increasing the hash table size (z3-after).

almost all the rejected formulas were turned into unknowns. Considering the results for CVC4/cvc5 (Figure 7.11 right), we observe almost no difference in the number of solved formulas except in the theory of Strings where many rejected formulas were solved from cvc4-1.7 to cvc4-1.8. Moreover, we observe a slight decrease in FP from cvc4-1.8 to the cvc5 versions. Strikingly in FP, all solvers of the CVC4/cvc5 family solve significantly fewer formulas than early Z3 releases do.

Understanding Z3's declining performance from z3-4.8.10 to z3-4.8.11

We analyzed the performance decline in Z3 from version 4.8.10 to version 4.8.11, the strongest effect we observed. Using bisection, we could pin it to the following root cause: In March 2021, a researcher observed a performance regression caused by hash collisions in Z3. He filed the following pull request:

```
 public:
+    ast_table() : chashtable({}, {}, 512 * 1024, 8 * 1024) {}
     void push_erase(ast * n);
     ast* pop_erase();
 };
```

src/ast/ast.h (Z3 #5040)

This increases the start size of Z3's hash table to 512 KB entries instead of 8 KB, the previous default size. As the researcher showed performance improvements for his application, Z3's lead developer merged the pull request into the trunk. Interestingly, another GitHub user reverted this change in his public fork. Furthermore, there was a discussion about the .NET API layer of Z3 related to this change. To understand its impact on larger formulas, we use FEAT's indexing feature. We extend ET in the following way: (1) we set the benchmarked solvers to the commits before and after the change, (2) we increase the variable count from two to five in each grammar, (3) we search for a start index *start_idx* for FEAT s.t. the corresponding formula is at least 4 KB, (4) we repeat 6 times:

a) Generate 5,000 formulas beginning at $FEAT(i)$

b) $i := i \cdot offset$

where the *offset* is 10^{10} and i is initially set to *start_idx* which varies from 10^{275} to $10^{1,700}$ depending on the grammar. Considering the results (Figure 7.12), we observe that the effect extends to larger formulas. In all of the six iterations, the Z3 version after the change (z3-after) solves significantly fewer formulas within 0.125 and 0.25 seconds. However, as we also observe, the effect is roughly constant vanishing after 0.125 seconds.

HIGHEST TIMEOUT T=8S For the highest timeout, we focus on the performance regressions which are the most interesting. Considering the results for Z3 (see Figure 7.13 left), we observe that Z3 solves a constant number of formulas since z3-4.8.10 and increases in earlier releases. Considering the results for CVC4/cvc5, we see two interesting effects. For BV, there is a decrease of over 5,000 formulas from cvc4-1.8 to cvc5-1.0.2, which is then rebounded in cvc5-1.0.3. The second interesting effect happens in Arrays from cvc5-1.0.5 to cvc5-1.0.6 with a drop of about 10,000 formulas.

CUMULATIVE RUNTIME & THROUGHPUT Besides the solved formulas, we consider cumulative runtime on jointly solved formulas to study the solvers' evolution on a set of fixed benchmarks and throughput as a practical metric for client software (see Figure 7.14). For the cumulative runtime on jointly solved formulas, the vertical axis unit is seconds, and for the throughput the unit is formulas per second. Let us first consider the runtime. As a general trend, Z3's runtime increases from 4.8.10 to 4.8.11 throughout all theories peaking at z3-4.12.2. Looking at CVC4/cvc5, we observe near-constant runtime in 7 out of 8 theories. The only exception is Bitvectors where there is fluctuation, with cvc4-1.7 being the fastest, an increase in runtime in early cvc5 releases, and a later decrease in cvc5-1.0.1.

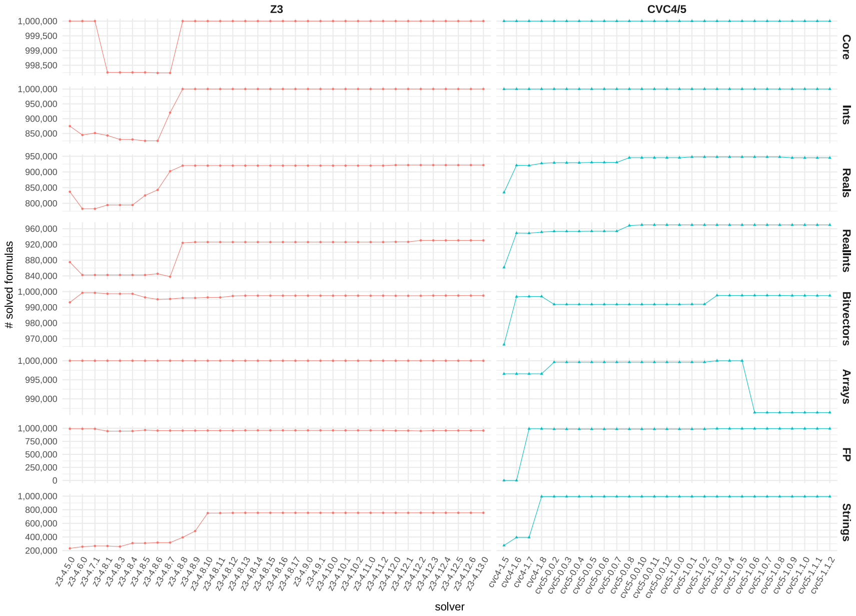


FIGURE 7.13: Number of solved formulas for the highest timeout T=8s.

Considering the throughput, we again observe the effect from 4.8.10 to 4.8.11, *i.e.*, a significant decrease in throughput. Besides this, we observe a fluctuation in Bitvectors. Considering CVC4/cvc5, we observe mild decreases in Core, and more significant decreases in Ints, Reals, and Strings. In Bitvectors and Arrays, cvc5-1.0.3 recovers from earlier drops.

Result #3: *Recent Z3 releases solve fewer formulas in all theories at the lowest timeout. At the highest timeout, early cvc5 versions solve fewer Bitvector than CVC4 and there are recent regressions in the theory of Arrays. Recent versions of both solvers have lower throughput than earlier releases.*

RQ4: How practical is ET for continous integration?

With the encouraging results from RQ1-RQ3, we next explore the practicality of ET for correctness and performance monitoring on commodity hardware. The full experiment with 61 SMT solvers and all eight theories (*i.e.*, 8 million formulas) takes about four days on a 96-core machine. However, for

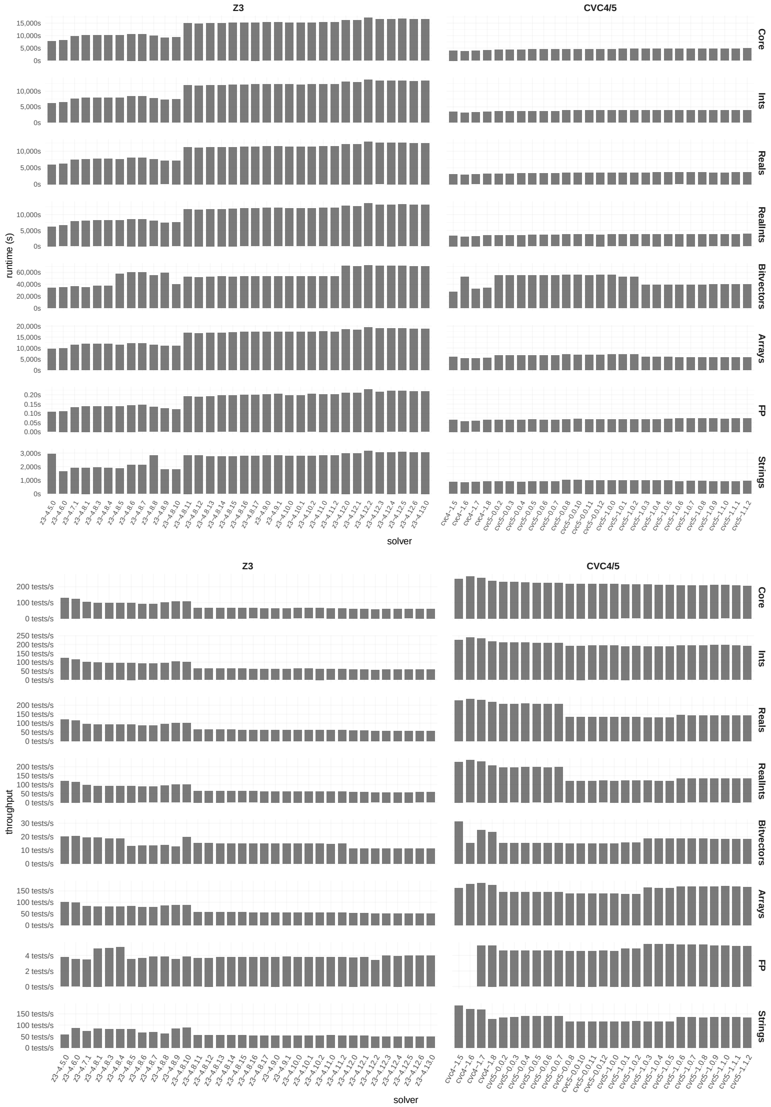


Figure 7.14: Top: Cumulative runtime on jointly solved formulas. Bottom: Throughput in formulas per second.

monitoring, we would have a different setup. We only need at most two SMT solvers, one to monitor and a reference solver. By caching the results of the reference solver, we can reduce to a single solver. To realize such a pipeline for Z3, we use the trunk for monitoring and the latest stable release of cvc5 as the reference solver and vice-versa for cvc5. We assume a CI/CD pipeline, *e.g.*, by GitHub actions, with two cores and a time limit of six hours per job. Investigating our data, we observe that 99% of bug triggers (from RQ2) are within the first 120,000 formulas, and 80% occur within the first 51,000 formulas. We further observe that 40% of the total time is spent on the FP theory. By exploiting these empirical facts, we can construct a pipeline by limiting the formula count to 51,000 (120,000) and excluding the FP theory. Feasible realizations take three hours and 23 minutes for Z3 to cover 80%, and 1h 18 minutes for cvc5 to cover 99% of the bugs.

Result #4: *ET is practical for continuous integration: for cvc5, we cover 99% of the bugs in less two hours, for Z3 we cover 80% of the bugs in three and a half hours on a commodity CI/CD pipeline.*

7.5 DISCUSSION

We first discuss ET's extensibility and scalability, then we discuss the importance of small formulas based on the small scope hypothesis, and finally we discuss ET's limitations.

EXTENSIBILITY AND SCALABILITY We evaluated ET on eight grammars derived from the official, quantifier-free SMT-LIB theories. However, we emphasize that ET is not restricted to these theories. A richer subset of SMT-LIB can be designed by extending the existing grammars or by designing new grammars. Additional operators and constants can be supported by adding them as terminals to the grammar, compiling, and re-running ET. Support for quantifiers, mixed theories, and incremental mode, can likewise be realized by modifications to expression productions in the grammars. As recent fuzzing campaigns reveal [2, 3, 20, 21], more than two-thirds of the bugs include quantifiers, incremental mode, tactics, and other non-standard features. The correctness results by ET can hence be thought of as a lower bound on the overall solver correctness. Enhancing the scalability of ET is possible by adding random sampling supported by the underlying

FEAT library. Conceptually, testing with a generic SMT-LIB grammar is also possible. However, such an approach suffers from combinatorics.

importance of small formulas ET is inspired by the small-scope hypothesis stating that most bugs have small triggers. While the hypothesis is known to hold for SMT solver correctness bugs, our findings suggest that it also does for performance bugs. Moreover, we believe that correctness and performance on small formulas are integral for establishing trust in SMT solvers. Little shakes users' trust in SMT solvers more than soundness bugs with small triggers. Similarly, performance regressions on small formulas undermine user's confidence in their performance. ET helps protect against these threats by detecting correctness and performance issues on small formulas before users report larger triggers. Small triggers are especially suited for triaging performance bugs for which reducers are often too aggressive resulting in excessive timeouts.

limitations ET is a powerful tester with many benefits. However, naturally, it also comes with its limitations. As ET has a differential oracle, it inherits the limitations of differential testing. E.g., the differential oracle could potentially miss soundness bugs if the reference and the tested solver both return the same incorrect result. To mitigate this, we enable the internal SMT solver's model validators and unsatisfiable core checks for satisfiable and unsatisfiable formulas, respectively. If a soundness bug is encountered, either procedure would halt the solver. Another limitation is that our results are subject to the variance of the machine. We hence repeated the experiments two times in an isolated setup, and disabled hyperthreading and frequency scaling.

7.6 related work

We first discuss related work on SMT solver robustness and performance testing, then enumerative and bounded exhaustive testing, and finally how our approach relates to benchmarking.

smt solver testing We found many correctness and performance bugs in Z3 and cvc5. Hence, ET is related to the family of correctness and performance testers for SMT solvers. Among the correctness testers, the first work is by Brummayer and Biere [2009] dating back almost 15 years. Their tool FuzzSMT found 16 solver defects in five older solvers and none in Z3.

Similar to ET, FuzzSMT is grammar-based and also on random generation. A later work was BtorMBT [61], a testing tool for Boolector [62], an SMT solver for the Bitvector theory. For almost a decade, soundness bugs in SMT solvers were rarely encountered and SMT solvers solidified greatly over the years, with Z3 and CVC4/cvc5 reaching industrial strength. Testing research at the time seemed to first confirm this [56, 63]. However, later STORM [32] and YinYang [1] found dozens of soundness bugs in Z3. Even later, OpFuzz, TypeFuzz and Falcon [2, 3, 20, 21] found several hundred bugs in Z3 and CVC4/cvc5. Besides the correctness fuzzers, StringFuzz [56] is the first performance tester. StringFuzz found two performance bugs in z3str3. Similar to ET, StringFuzz targets both correctness and performance bugs and is based on grammar. A follow-up work is BanditFuzz [58] which guides the testing by reinforcement learning. As a key difference to all correctness and performance testers, ET's testing is systematic and enumerative rather than unsystematic and random.

ENUMERATIVE TESTING AND BOUNDED EXHAUSTIVE TESTING ET is based on the functional enumerator FEAT [33], which belongs to the family of property-based testers. Another similar tool is LeanCheck [114] supporting richer properties than FEAT. ET is loosely related to the enumerative tester SmallCheck [115], and the random property-based tester QuickCheck [116]. In the software engineering community, researchers proposed Bounded Exhaustive Testing, through an approach for testing Galileo, a dynamic fault tree analysis tool [117]. Similar to our work, their approach enumerates inputs "to improve the assurance levels of complex software", however different from our approach, they use the analyzer Alloy [118] for input generation. More loosely related is skeletal program enumeration (SPE) [82], an approach for validating compilers. Different from the enumerative testing, SPE does not fully enumerate the input space. Instead, it uses existing inputs to generate holes and then fills those holes with type-conforming terms. As ET generates tests from context-free grammars, grammar-based black-box fuzzers [91, 102, 119, 120] are related. Unlike grammar-based fuzzers, ET is size-bounded and enumerative rather than depth-bounded and random.

BENCHMARKING Our study on the evolution of SMT solvers is related to benchmarking. The most prominent benchmarking initiative is SPEC [121], which regularly evaluates hardware, software and systems on a large set of real-world benchmarks. SPEC's benchmarking includes CPU performance, cloud-computing, Java environments, *e.g.*, SPEC Java. Another Java benchmark is DaCapo [122], a diverse client-side benchmark suite with large-scale applications such as ANTLR, hqldb, eclipse, and jython. A different strand of benchmarking initiatives are solver competitions in automated reasoning and operations research [123, 124, 125]. Most closely related is the SMT-COMP [126], the yearly SMT solver competition. In the SMT-COMP competition, SMT solvers compete in several categories on the SMT-LIB benchmark set plus additional benchmarks supplied by users of SMT solvers. Most of the SMT-LIB benchmarks consist of applications that are intentionally challenging for SMT solvers. As a key difference from all three benchmarks SMT-LIB, SPEC, and DaCapo, which measure the performance on real-world applications, ET's formulas are enumerative and not based on applications. Moreover, ET assesses the correctness of SMT solvers besides performance, complementing SMT-COMP in ensuring SMT solver's correctness and performance.

8 CONCLUSION AND OUTLOOK

The overarching goal of this thesis was to solidify modern SMT solvers. This chapter first provides a summary of contributions, *i.e.*, the proposed methodologies, approaches, and tools. Then we discuss the impact of these contributions. We close with an outlook on future work.

8.1 SUMMARY

The first work *Semantic Fusion* put the existing trust in modern SMT solvers to the test. Its results showed that SMT solvers are less stable than previously believed. Semantic fusion is a generic methodology that fuses two test cases into a new test case with a known oracle. It belongs to the family of metamorphic testing techniques and can be employed with a single SMT solver. Complementing Semantic Fusion, we devised *Type-Aware Operator Mutation*, a testing technique with a differential oracle. Its key idea is to mutate the operators within SMT formulas by other operators of the same type. The technique is simple but unusually effective, finding 1,254 bugs in Z3 and CVC4. Despite its effectiveness, Type-Aware Operator Mutation has a key shortcoming: it cannot grow or shrink SMT formulas. To overcome this, we devised an even more powerful technique: *Generative Type-Aware Mutation* which can also generate expressions with fresh operators and then replace existing expressions in the formula. The technique found another 322 bugs in Z3 and CVC4/cvc5 among them many critical soundness bugs.

While the first three works focus on correctness, for performance, we devised Janus, an approach for finding incompleteness bugs in SMT solvers. The key insight is realized in a technique called *Weakening and Strengthening*. In a nutshell: to mutate SMT formulas with local implication rules. Janus is effective: we found 31 incompleteness bugs, 26 have been confirmed, and 20 are already fixed. Janus' diverse bugs uncovered functional, regression, and performance bugs; several triggered discussions among the developers.

Over the years, our and other bug-hunting campaigns have led to several hundreds of bug fixes in the SMT solvers Z3 and CVC4/cvc5. However, despite these efforts, all existing testers were unsystematic, i.e. not yielding any guarantees and missing bugs with small triggers. To tackle this, we devised ET, a tool implementing *Grammar-based Enumeration* for validating SMT solver correctness and performance. Despite the extensive and continuous testing of the Z3 and cvc5, ET found bugs, out of which 76 were confirmed and 32 were fixed. Moreover, ET can help understand the evolution of solvers. We derived eight grammars realizing all major SMT theories including the booleans, integers, reals, realints, bit-vectors, arrays, floating points, and strings. Using ET, we test all consecutive releases of the SMT solvers Z3 and CVC4/cvc5 from the last six years. Our results suggest improved correctness in recent versions of both solvers but decreased performance in newer releases of Z3 on small timeouts (since z3-4.8.11) and regressions in early cvc5 releases on larger timeouts.

8.2 IMPACT

IMPACT ON SMT SOLVERS This thesis enabled one of the world's largest academic bug-hunting campaigns. Given this major effort, we ask the following question: *Did SMT solvers solidify through this work?* We observe that *developers fixed a substantial amount of the reported bugs:* 1,333 *of all bugs and 349 of all soundness bugs in Z3 and CVC4/cvc5.* [1] These fixes not only included soundness bugs that were undetected for years but also prevented new soundness bugs from being introduced to solver releases. One striking example is z3-4.8.8, where our bug hunting prevented more than 23 soundness bugs alone from manifesting in the release.[2] All other releases of Z3 and CVC4/cvc5 since z3-4.8.7 and cvc5-1.0.2 likewise greatly benefited from our bug reports. Empowered by ET (see Chapter 7), we generalize the question. *Did SMT solvers solidify through automated testing?* Besides the our findings, we now have to include fixes caused by other testing campaigns [20, 21, 30, 32].[3] Using ET's results, we can partially approach the question for quantifier-free SMT theories (see Figure 8.1). The highlighted areas in the respective subplots show the releases subject to testing campaigns. As we can see: since the beginning of the fuzzing campaigns, the bug counts have decreased significantly and so have the number of theories with bugs. Moreover, beginning z3-4.8.8 and cvc5-0.0.8 no soundness bugs were found by ET's systematic enumeration. *We hence argue that this thesis has significantly solidified SMT solvers.*

[1] Many of the unfixed bugs are still pending and might also be fixed shortly.

[2] https://github.com/Z3Prover/z3/releases/tag/z3-4.8.8

[3] At the time of writing, our work amounts to more than half the bug fixes in Z3 and CVC4/cvc5.

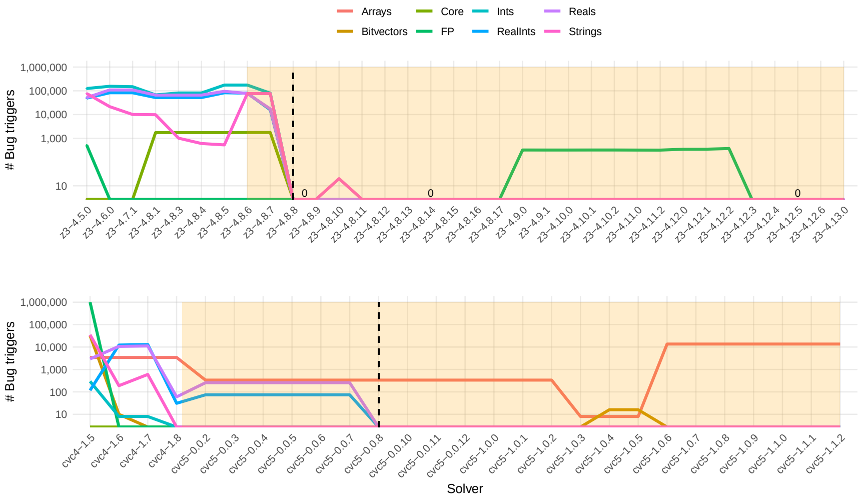


FIGURE 8.1: Bug triggers in releases of Z3 (top) and CVC4/cvc5 (bottom). The highlighted area indicates solver releases subject to testing campaigns. To the right of the dashed line, releases exhibit no soundness bugs.

FURTHER IMPACT Besides the bug-hunting results, this thesis established a new research (sub)-area with follow-up works within and outside the SMT community. Works divide roughly into SMT testers [19, 20, 21, 22, 23, 24, 25, 30], testers for other formal methods tools [26, 27, 28] and general software testing works [19, 29]. Our tools YinYang and Janus, are already open-sourced. ET will be open-sourced soon. YinYang and Janus have a combined 190 stars and 22 forks on GitHub. Researchers and practitioners can benefit from our tools, i.e. making their SMT solvers robust or judging the robustness of a given SMT solver. Our research also has significant recognition in industry. E.g., our tool YinYang was awarded a Google Open-Source Peer bonus in 2021 and an Amazon Research Award in 2022. During my internship at AWS in 2023, I applied ET to the string solver nfa2sat which is used for Zelkova in production. ET is part of the continuous integration tests for nfa2sat [31].

8.3 OUTLOOK

Our research line has opened up an exciting body of follow-up work. We outline three strands of future work to further improve the quality of SMT solvers and generalize our testing techniques.

DIRECTED TESTING OF SMT SOLVER & PROOF MODES Because of the complexity of Z3 and CVC4/cvc5's code (looping, and recursion), coverage-guided testing of the code of the SMT solvers Z3 and CVC4/cvc5 has not demonstrated to be effective. [4] However, instrumenting only parts of the code, could make coverage-guided testing effective. One idea is the focused testing of a single component such as the rewriter or proof mode. As bugs in rewriters have been the main source for soundness issues in SMT solvers, this could further solidify the solvers.

AUTOMATED BUG FIXING & DEBLOATING OF SMT SOLVERS Automated fixing of SMT solver bugs could complement automated bug-hunting. The high-level idea would be to use machine-learning-based techniques from program repair such as getafix [127]. We then aim to couple them with fuzzing techniques for SMT solvers as oracles to sort out the various fixes. Another approach to improve SMT solvers is by debloating code bases of Z3 and cvc5 through data-driven analysis. Using the SMT benchmarks, it could be possible to identify rarely-used optimizing components in the solvers (*e.g.*, rewriters). The goal is then to reduce the solvers to the bare minimum while maintaining performance.

GENERALIZATION OF THE TESTING TECHNIQUES TO OTHER DOMAINS Another strand of future work could be the generalization of our testing techniques to other domains. The most direct candidate for generalization is the grammar-based enumerator ET. We plan to devise grammars for JavaScript, Golang, and the Solidity languages. Large-scale generalization of Generative Type-Aware Mutation could be possible by letting LLMs label the types of input seeds and then mutating the typed seeds. A differential oracle between the latest release of software and the trunk could help detect many bugs. An interesting challenge would be the extraction of the bugs among all the inconsistencies between the latest release and trunk. Semantic Fusion is suitable for domains where differential testing is inapplicable.

[4] We tried enhancing OpFuzz with an Superion [89] to benefit from coverage guidance.

Besides SMT solvers, it has been applied to validate multimedia content moderation software [128].

More broadly, this thesis enables *Formal Method Engineering*, a new engineering discipline to make formal methods more reliable, performant, and usable. [5] As we demonstrate for SMT solvers, automated software testing is a powerful tool for ensuring reliability and performance. This yields a positive perspective on ensuring the reliability and performance of formal methods tools beyond SMT solvers.

[5] The idea of a dedicated engineering discipline for formal methods was brought up by Peter Müller in a joint discussion.

BIBLIOGRAPHY

[1] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Validating SMT Solvers via Semantic Fusion”. In: *PLDI '20*. 2020, 718.

[2] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “On the Unusal Effectiveness of Type-Aware Operator Mutation”. In: *OOPSLA '20*. 2020, 1.

[3] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Generative Type-Aware Mutation for Testing SMT Solvers”. In: *OOPSLA '21*. 2021, 1.

[4] Mauro Bringolf, Dominik Winterer, and Zhendong Su. “Finding and Understanding Incompleteness Bugs in SMT Solvers”. In: *ASE '22*. 2022, 1.

[5] Dominik Winterer. “Grammar-based Enumeration of SMT Solvers for Correctness and Performance”. In: *OOPSLA '24*. 2024.

[6] Vince Szabo, Dominik Winterer, and Zhendong Su. “Compilation Quotient (CQ): A Metric for the Compilation Hardness of Programming Languages”. In: *arXiv*. 2024.

[7] National Institute of Standards and Technology (NIST). *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Tech. rep. U.S. Department of Commerce, 2024.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *PLDI '05*. 2005, 213.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *OSDI '08*. 2008, 209.

[10] Armando Solar-Lezama. “Program Synthesis by Sketching”. PhD thesis. University of California at Berkeley, 2008.

[11] Emina Torlak and Rastislav Bodik. “A lightweight symbolic virtual machine for solver-aided host languages”. In: *PLDI '14*. 2014, 530.

[12] David Detlefs, Greg Nelson, and James B. Saxe. “Simplify: A Theorem Prover for Program Checking”. In: *JACM* (2005), 365.

[13] Rob DeLine and Rustan Leino. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Tech. rep. 2005.

[14] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. "CBMC: The C Bounded Model Checker". In: *arXiv*. 2023.

[15] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *CAV '17*. 2017, 97.

[16] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *FMCAD '18*. 2018, 1.

[17] AdaCore. *SPARK*. URL: https://github.com/AdaCore/spark2014 (last visited on 2024-07-08).

[18] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. "Automated Analysis and Debugging of Network Connectivity Policies". In: MSR-TR-2014-102 (2014).

[19] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. "Fuzz4all: Universal fuzzing with large language models". In: *ICSE '24*. 2024, 1.

[20] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. "Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration". In: *ISSTA '21*. 2021, 322.

[21] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. "Skeletal Approximation Enumeration for SMT Solver Testing". In: FSE '21. 2021, 1141.

[22] Hichem Rami Ait El Hara, Guillaume Bury, and Steven de Oliveira. "Alt-Ergo-Fuzz: A fuzzer for the Alt-Ergo SMT solver". In: *33èmes Journées Francophones des Langages Applicatifs*. 2022, 235.

[23] Anzhela Sukhanova and Valentyn Sobol. "HornFuzz: Fuzzing CHC solvers". In: *EASE '23*. 2023, 83.

[24] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. "Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs". In: *ICSE '23*. 2023, 69.

[25] Jongwook Kim, Sunbeom So, and Hakjoo Oh. “Diver: Oracle-Guided SMT Solver Testing with Unrestricted Random Mutations”. In: *ICSE '23*. 2023, 2224.

[26] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. “Metamorphic testing of Datalog engines”. In: *FSE '21*. 2021, 639.

[27] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. “Dependency-Aware Metamorphic Testing of Datalog Engines”. In: *ISSTA '23*. 2023, 236.

[28] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. “Testing Dafny (experience paper)”. In: *ISSTA '22*. 2022, 556.

[29] Andrei Lascu, Alastair F. Donaldson, Tobias Grosser, and Torsten Hoefler. “Metamorphic Fuzzing of C++ Libraries”. In: *ICST '22*. 2022, 35.

[30] Aina Niemetz, Mathias Preiner, and Clark Barrett. “Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers”. In: *CAV '22*. 2022, 92.

[31] Kevin Lotz, Amit Goel, Bruno Dutertre, Benjamin Kiesl-Reiter, Soonho Kong, Rupak Majumdar, and Dirk Nowotka. “Solving string constraints using SAT”. In: *CAV '23*. 2023.

[32] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. “Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing”. In: *FSE '20*. 2020, 701.

[33] Jonas Duregard, Patrick Jansson, and Meng Wang. “FEAT: Functional Enumeration of Algebraic Types”. In: *Haskell '12*. 2012, 61.

[34] D. Jackson and C.A. Damon. “Elements of style: analyzing a software design feature with a counterexample detector”. In: *IEEE Transactions on Software Engineering* (1996), 484.

[35] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. *Evaluating the “Small Scope Hypothesis”*. Tech. rep. 2002.

[36] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 2nd ed. Springer Publishing Company, Incorporated, 2008.

[37] Greg Nelson and Derek C. Oppen. “Simplification by Cooperating Decision Procedures”. In: *TOPLAS* (1979), 245.

[38] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS '08*. 2008, 337.

[39] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *CAV '11*. 2011, 171.

[40] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *TACAS '22*. 2022, 415.

[41] Bruno Dutertre and Leonardo De Moura. *The Yices SMT Solver*. Tech. rep. 2006.

[42] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *TACAS '13*. 2013, 93.

[43] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. "The OpenSMT Solver". In: *TACAS*. 2010, 150.

[44] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: an interpolating SMT solver". In: *SPIN'12*. 2012, 248.

[45] Sylvain Conchon, Mohamed Iguernlala, David Mentré, Guillaume Melquiond, and Virginie Wiels. *Alt-Ergo: The SMT Solver*. URL: https://ocamlpro.github.io/alt-ergo (last visited on 2024-07-08).

[46] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *CAV '07*. 2007, 519.

[47] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*.

[48] Cristian Cadar and Alastair Donaldson. "Analysing the Program Analyser". In: *ICSE '16*. 2016, 765.

[49] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. "MaxSMT-Based Type Inference for Python 3". In: *CAV '18*. 2018, 12.

[50] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers". In: *Automated Deduction – CADE-23*. 2011, 116.

[51] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq”. In: *CAV ’17*. 2017, 126.

[52] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft.” In: *Queue* (2012), 20.

[53] Neha Rungta. “A Billion SMT Queries a Day (Invited Paper)”. In: *CAV ’22*. 2022, 3.

[54] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. “Test-case reduction for C compiler bugs”. In: *PLDI ’12*. 2012, 335.

[55] Gereon Kremer. *pyDelta: delta debugging for SMT-LIB*. URL: https://github.com/nafur/pydelta (last visited on 2024-07-08).

[56] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. “StringFuzz: A Fuzzer for String Solvers”. In: *CAV ’18*. 2018, 45.

[57] *Using the GNU Compiler Collection (GCC): Gcov*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (last visited on 2024-07-08).

[58] Joseph Scott, Federico Mora, and Vijay Ganesh. “BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning”. In: *CAV ’20*. 2020, 68.

[59] Celeb Stanford, Margus Veanes, and Nikolaj Bjørner. “Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints”. In: *PLDI ’21*. 2021, 620.

[60] Robert Brummayer and Armin Biere. “Fuzzing and delta-debugging SMT solvers”. In: *SMT ’09*. 2009, 1.

[61] Aina Niemetz, Mathias Preiner, and Armin Biere. “Model-based API testing for SMT solvers”. In: *SMT ’17*. 2017, 10.

[62] Robert Brummayer and Armin Biere. “Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays”. In: *TACAS ’09*. 2009, 174.

[63] Alexandra Bugariu and Peter Müller. “Automatically Testing String Solvers”. In: *ICSE ’20*. 2020, 1459.

[64] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. “Finding and understanding bugs in software model checkers”. In: *FSE ’19*. 2019, 763.

[65] Christian Klinger, Maria Christakis, and Valentin Wüstholz. "Differentially testing soundness and precision of program analyzers". In: *ISSTA '19*. 2019, 239.

[66] Timotej Kapus and Cristian Cadar. "Automatic testing of symbolic execution engines via program generation and differential testing". In: *ASE '17*. 2017, 590.

[67] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. "Effective dynamic detection of alias analysis errors". In: *FSE '13*. 2013, 279.

[68] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. "Automatically testing implementations of numerical abstract domains". In: *ASE '18*. 2018, 768.

[69] Lina Qiu, Yingying Wang, and Julia Rubin. "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe". In: *ISSTA '18*. 2018, 176.

[70] Felix Pauck, Eric Bodden, and Heike Wehrheim. "Do android taint analysis tools keep their promises?" In: *FSE '18*. 2018, 331.

[71] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. *Metamorphic testing: a new approach for generating next test cases*. Tech. rep. 1998.

[72] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs". In: *PLDI '14*. 2014, 216.

[73] Vu Le, Chengnian Sun, and Zhendong Su. "Finding deep compiler bugs via guided stochastic program mutation". In: *OOPSLA '15*. 2015, 386.

[74] Chengnian Sun, Vu Le, and Zhendong Su. "Finding compiler bugs via live code mutation". In: *OOPSLA '16*. 2016, 849.

[75] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. "Many-core compiler fuzzing". In: *PLDI '15*. 2015, 65.

[76] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. "Automated testing of graphics shader compilers". In: *OOPSLA '17*. 2017, 93.

[77] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. "Z3str3: A String Solver with Theory-aware Branching". In: *FMCAD '17*. 2017, 55.

[78] Z3. *Z3 Regression Test Suite*. URL: https://github.com/Z3Prover/z3test (last visited on 2024-07-04).

[79] CVC4. *CVC4 Regression Test Suite*. URL: https://github.com/CVC4/CVC4/tree/master/test/regress (last visited on 2024-04-29).

[80] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Clark W. Barrett, and Cesare Tinelli. “On Counterexample Guided Quantifier Instantiation for Synthesis in CVC4”. In: *CAV ’15*. 2015.

[81] Aina Niemetz and Armin Biere. “ddSMT: A Delta Debugger for the SMT-LIB v2 Format”. In: *SMT ’13*. 2013, 36.

[82] Qirun Zhang, Chengnian Sun, and Zhendong Su. “Skeletal program enumeration for rigorous compiler testing”. In: *PLDI ’17*. 2017, 347.

[83] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. “Coverage guided, property based testing”. In: *OOPSLA ’19*. 2019, 1.

[84] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-adaptive mutational fuzzing”. In: *SP ’15*. 2015, 725.

[85] Michal Zalewski. *american fuzzy lop*.

[86] Caroline Lemieux and Koushik Sen. “Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage”. In: *ASE ’18*. 2018, 475.

[87] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. “Steelix: Program-state based binary fuzzing”. In: *FSE ’17*. 2017, 627.

[88] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. “Smart greybox fuzzing”. In: *TSE ’19*, 1980.

[89] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Superion: grammar-aware greybox fuzzing”. In: *ICSE ’19*. 2019, 724.

[90] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. “NAUTILUS: Fishing for Deep Bugs with Grammars”. In: *NDSS ’19*. 2019, 337.

[91] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *PLDI ’11*. 2011, 283.

[92] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. “Compiler fuzzing through deep learning”. In: *ISSTA ’18*. 2018, 95.

[93] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. “APOLLO: Automatic detection and diagnosis of performance regressions in database systems”. In: *VLDB ’19*. 2019, 57.

[94] Andreas Seltenreich. *SQLSmith*. https://github.com/anse1/sqlsmith (last visited on 2024-07-04).

[95] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. “Generating targeted queries for database testing”. In: *SIGMOD ’08*. 2008, 499.

[96] Manuel Rigger and Zhendong Su. “Detecting optimization bugs in database engines via non-optimizing reference Engine Construction”. In: *OOPSLA ’20*. 2020, 1140.

[97] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. “Difuze: Interface aware fuzzing for kernel drivers”. In: *CCS ’17*. 2017, 2123.

[98] HyungSeok Han and Sang Kil Cha. “Imf: Inferred model-based fuzzer”. In: *CCS ’17*. 2017, 2345.

[99] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. “kAFL: Hardware-assisted feedback fuzzing for OS kernels”. In: *USENIX Security ’17*. 2017, 167.

[100] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. “Exploiting the Saturation Effect in Automatic Random Testing of Android Applications”. In: *MOBILESoft ’15*. 2015, 33.

[101] Dominik Winterer, Chengyu Zhang, and Zhendong Su. *yinyang: a fuzzer for SMT solvers*. URL: `https://github.com/testsmt/yinyang` (last visited on 2024-07-04).

[102] K. V. Hanford. “Automatic generation of test cases”. In: *IBM Systems Journal* 9.4 (1970), 242.

[103] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. “Random Testing for C and C++ Compilers with YARPGen”. In: *OOPSLA ’20*. 2020, 1.

[104] Manuel Rigger and Zhendong Su. “Testing Database Engines via Pivoted Query Synthesis”. In: *OSDI ’20*. 2020, 667.

[105] Patrice Godefroid, Hila Peleg, and Rishabh Singh. “Learn Fuzz: Machine learning for input fuzzing”. In: *ASE ’17*. 2017, 50.

[106] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. “Provably correct peephole optimizations with alive”. In: *PLDI ’15*. 2015, 22.

[107] Levent Erkok. *SBV: SMT Based Verification in Haskell*. URL: `https://github.com/LeventErkok/sbv` (last visited on 2024-07-08).

[108] Zvonimir Rakamaric and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *CAV '14*. 2014, 106.

[109] Rustan Leino and Clément Pit-Claudel. “Trigger Selection Strategies to Stabilize Program Verifiers”. In: *CAV '16*. 2016.

[110] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. “Mariposa: Measuring SMT Instability in Automated Program Verification”. In: *FMCAD '23*. 2023, 178.

[111] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. “SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities”. In: *CCS '17*. 2017, 2155.

[112] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. “PerfFuzz: Automatically Generating Pathological Inputs”. In: *ISSTA '18*. 2018, 254.

[113] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. “Automating Performance Bottleneck Detection Using Search-Based Application Profiling”. In: *ISSTA '15*. 2015, 270.

[114] Rudy Matela Braquehais. “Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing”. PhD thesis. University of York, 2017.

[115] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. “Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values”. In: *Haskell '08*. 2008, 37.

[116] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: ICFP '00. 2000, 268.

[117] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. “Software Assurance by Bounded Exhaustive Testing”. In: *ISSTA '04*. 2004, 133.

[118] Daniel Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Trans. Softw. Eng. Methodol.* (2002).

[119] W.H. Burkhardt. “Generating test programs from syntax”. In: *Computing*. 1967, 53.

[120] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *USENIX Security '12*. 2012, 38.

[121] SPEC. *SPEC's Benchmarks and Tools*. `https://www.spec.org/benchmarks.html` (last visited on 2024-07-04).

[122] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06*. 2006, 169.

[123] SAT Competition. *The International SAT Competition Web Page*. URL: `https://github.com/AdaCore/spark2014` (last visited on 2024-07-08).

[124] Geoff Sutcliffe. "The CADE ATP System Competition — CASC". In: *AI Magazine* (2016), 99.

[125] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library". In: *Mathematical Programming Computation* (2021).

[126] Clark Barrett, Leonardo de Moura, and Aaron Stump. "SMT-COMP: Satisfiability Modulo Theories Competition". In: *CAV '05*. 2005, 20.

[127] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. "Getafix: learning to fix bugs automatically". In: OOPSLA '19. 2019, 1.

[128] Wenxuan Wang, Jingyuan Huang, Chang Chen, Jiazhen Gu, Jianping Zhang, Weibin Wu, Pinjia He, and Michael Lyu. "Validating Multimedia Content Moderation Software via Semantic Fusion". In: *ISSTA '23*. 2023, 576.

A

APPENDIX

A.1 SEMANTIC FUSION

Proposition 3.2.1 (Mixed fusion). *Let* φ_1 *be a satisfiable and* φ_2 *be an unsatisfiable formula with* $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. *Let further* $x \in vars(\varphi_1)$, $y \in vars(\varphi_2)$ *be variables. Then:*

$$\varphi_{mixed\text{-}sat} = \varphi_1[r_x(y,z)/x] \vee \varphi_2[r_y(x,z)/y] \text{ is satisfiable; and}$$
$$\varphi_{mixed\text{-}unsat} = \varphi_1[r_x(x,z)/x] \wedge \varphi_2[r_y(x,z)/y] \;\wedge z = g(x,y) \text{ is unsatisfiable.}$$

Proof. For the satisfiability of $\varphi_{\text{mixed-sat}}$, we consider the proof for SAT fusion (Proposition 3.2.1) where we construct a model for φ_{sat} by structural induction. As an intermediate step, this proof concludes that $\varphi_1[r_x(y,z)/x]$ is satisfiable for a satisfiable φ_1. This also holds for $\varphi_{\text{mixed-sat}}$. Since $\varphi_1[r_x(y,z)/x]$ is satisfiable, so is $\varphi_{\text{mixed-sat}}$ since $\varphi_2[r_y(x,z)/y]$ is connected via disjunction and may be unsatisfiable but $\varphi_{\text{mixed-sat}}$ is still satisfiable. For the unsatisfiability of $\varphi_{\text{mixed-unsat}}$, we apply the following reasoning. Since φ_1 is satisfiable, $\varphi_1[r_x(y,z)/x]$ is satisfiable, as proved above. Hence, for $\varphi_{\text{mixed-unsat}}$ to contradict the assumption (that is to be satisfiable), $\varphi_2[r_y(x,z)/y] \wedge z = g(x,y)$ must be satisfiable. However, then we apply the same reasoning as in the Proof of Proposition 3.2.2 and show by contradiction that $\varphi_2[r_y(x,z)/y] \wedge z = g(x,y)$ is unsatisfiable for an unsatisfiable φ_2. Consequently, $\varphi_{\text{mixed-unsat}}$ is unsatisfiable as $\varphi_2[r_y(x,z)/y] \wedge z = g(x,y)$ is. □

A.2 TYPE-AWARE OPERATOR MUTATION

Logic	# non-inc	# inc	# total
QF_SLIA	67,584	-	67,584
QF_FP	40,318	2	40,320
QF_NIA	23,876	10	23,886
AUFLIRA	20,011	-	20,011
QF_ABVFP	18,093	69	18,162
QF_BVFP	17,231	182	17,413
QF_ABV	15,084	1,272	16,356
UFNIA	13,509	-	13,509
QF_NRA	11,489	-	11,489
UFLIA	10,137	-	10,137
QF_UF	7,457	766	8,223
QF_DT	8,000	-	8,000
UF	7,668	-	7,668
QF_LIA	6,947	69	7,016
BV	5,846	18	5,864
UFDT	4,527	-	4,527
NRA	3,813	-	3,813
QF_UFBV	1,234	2,330	3,564
AUFLIA	3,276	-	3,276
FP	2,484	-	2,484
LRA	2,419	5	2,424
QF_S	2,319	-	2,319
QF_IDL	2,193	-	2,193
UFLRA	15	1,870	1,885
AUFBVDTLIA	1,708	-	1,708
QF_LRA	1,648	10	1,658
AUFNIRA	1,480	165	1,645
QF_AUFLIA	1,303	72	1,375
QF_UFLIA	583	773	1,356
QF_UFLRA	1,284	-	1,284
AUFDTLIA	728	-	728
LIA	607	6	613
QF_AX	551	-	551
QF_UFNIA	478	1	479
QF_UFIDL	428	-	428
UFDTLIA	327	-	327
QF_RDL	255	-	255
BVFP	224	10	234
QF_ALIA	126	44	170
QF_BVFPLRA	168	-	168
UFBV	121	-	121
QF_ANIA	95	5	100
QF_AUFBV	56	31	87
UFIDL	68	-	68
ALIA	42	24	66
QF_FPLRA	57	-	57
QF_UFNRA	37	-	37
ABVFP	30	4	34
NIA	20	-	20
QF_AUFNIA	17	-	17
QF_LIRA	7	-	7
AUFNIA	3	-	3
QF_NIRA	3	-	3
UFDTNIA	1	-	1
cvc4regr	176	1,594	1,770
z3test	479	841	1,320
Total	308,640	10,173	318,813

Figure A.1: Formula counts for the respective benchmark sets. Column #non-inc refers to the count of non-incremental SMT-LIB files, column #inc refers to the count of incremental SMT-LIB files. *z3test* and *cvc4regr* refer to CVC4's and Z3's respective regression test suites.

A.3 WEAKENING AND STRENGTHENING/JANUS

Lemma 6.3.1. *Let φ be a formula with a subformula F. For any G weaker than F, we have:*

$$\begin{array}{ll} \textit{if } F \textit{ positive in } \varphi \textit{ then} & \forall x_1, \ldots, x_n \colon \varphi \to \varphi[F \mapsto G] \\ \textit{if } F \textit{ negative in } \varphi \textit{ then} & \forall x_1, \ldots, x_n \colon \varphi[F \mapsto G] \to \varphi \end{array}$$

with the set of free variables $FV(\varphi) = x_1, \ldots, x_n$.

Proof. By induction over φ. For every case, we only consider F being positive in φ as the negative cases are symmetric.

Case $\varphi = F$: direct.

Case $\varphi = \neg\varphi_1$: Let $x_1, \ldots, x_n = FV(\varphi) = FV(\varphi_1)$. Say *parity*$(F, \varphi) = 1$, then *parity*$(F, \varphi_1) = -1$ and by induction hypothesis for φ_1:

$$\forall x_1, \ldots, x_n \colon \varphi_1[F \mapsto G] \to \varphi_1$$

By contraposition, we obtain the opposite implication for φ.

Case $\varphi = \varphi_1 \wedge \varphi_2$: Let $x_1, \ldots, x_n = FV(\varphi)$. Without loss of generality, assume F is a subformula of φ_1 and $FV(\varphi_1) = x_1, \ldots, x_k$ with $k \leq n$. Consider *parity*$(F, \varphi) = 1$, then *parity*$(F, \varphi_1) = 1$ and by induction hypothesis for φ_1:

$$\forall x_1, \ldots, x_k \colon \varphi_1 \to \varphi_1[F \mapsto G]$$

Since $x_{k+1}, \ldots, x_n \notin FV(\varphi_1)$ this extends to:

$$\forall x_1, \ldots, x_n \colon \varphi_1 \wedge \varphi_2 \to (\varphi_1 \wedge \varphi_2)[F \mapsto G]$$

Note that because F is a subtree of the abstract syntax tree of φ_1 the substitution $[F \mapsto G]$ has no effect when applied to φ_2.

Case $\varphi = \exists x \colon \varphi_1$: Assume *parity*$(F, \varphi) = 1$, then *parity*$(F, \varphi_1) = 1$ and by induction hypothesis for φ_1:

$$\forall x_1, \ldots, x_n \colon \varphi_1 \to \varphi_1[F \mapsto G]$$

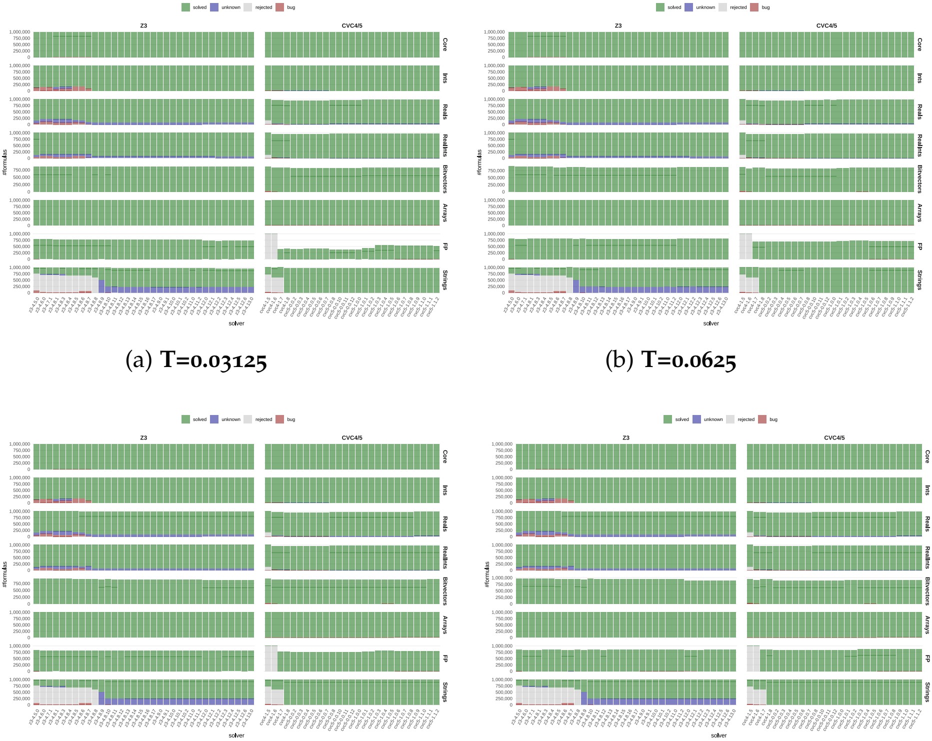
where $x_1, \ldots, x_n = FV(\varphi_1)$. If x does not occur free in φ_1, then $FV(\varphi) = FV(\varphi_1)$ and we can directly conclude the same implication for φ. Otherwise $x \in FV(\varphi_1)$ and without loss of generality $x = x_n$. Then the induction hypothesis implies:

$$
\begin{aligned}
&\forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \to (\exists x_n \colon \varphi_1[F \mapsto G]) \\
\Longleftrightarrow &\forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \to ((\exists x_n \colon \varphi_1)[F \mapsto G])
\end{aligned}
$$

Since $FV(\varphi) = FV(\varphi_1) - \{x\}$, concluding the proof. □

A.4 GRAMMAR-BASED ENUMERATION/ET

Number of solved formulas (bar plots)

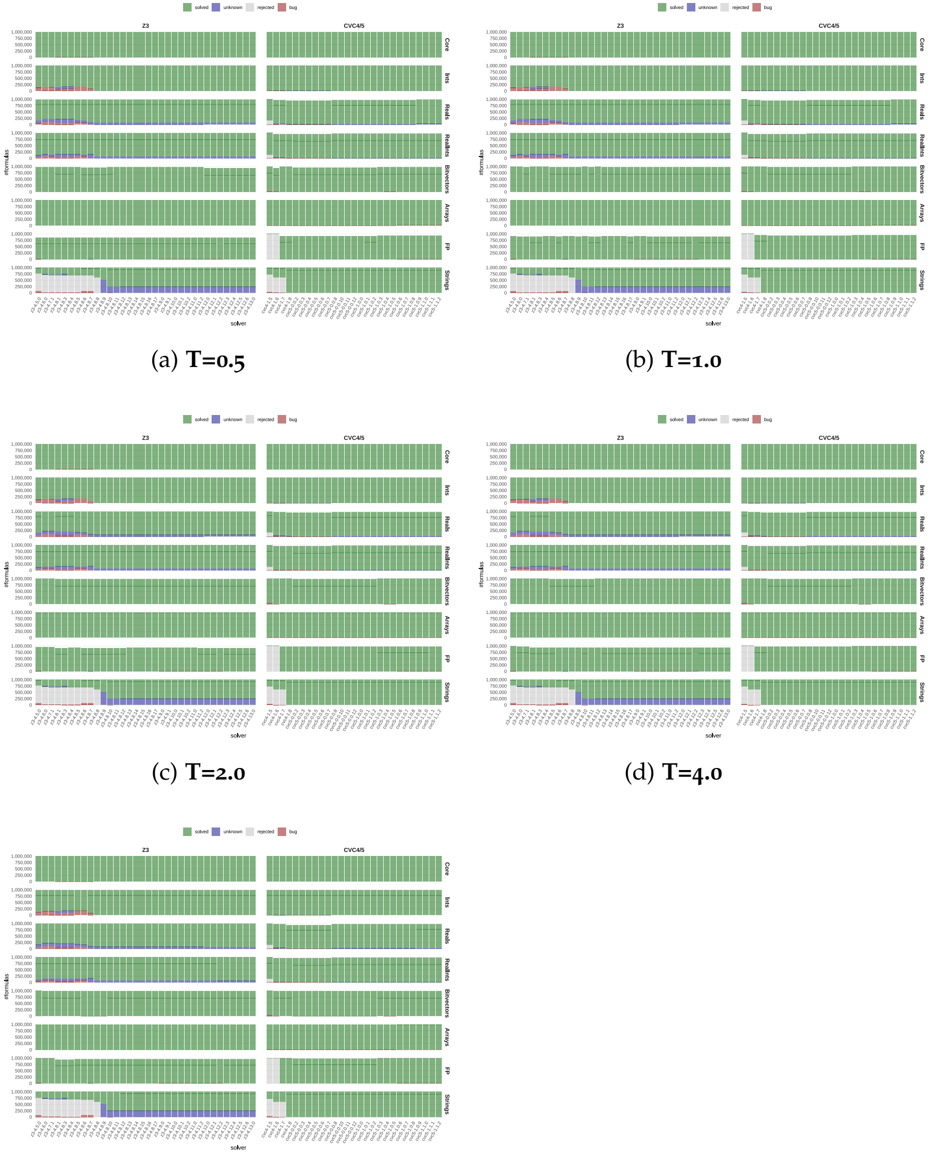


(a) **T=0.03125**

(b) **T=0.0625**

(c) **T=0.125**

(d) **T=0.25**



(a) **T=0.5**

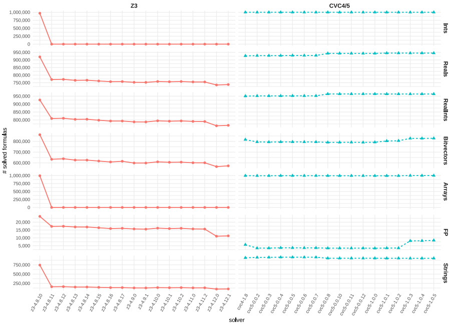
(b) **T=1.0**

(c) **T=2.0**

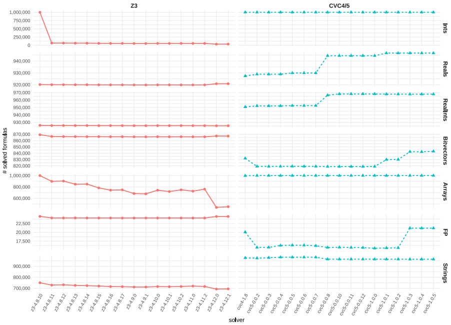
(d) **T=4.0**

(e) **T=8.0**

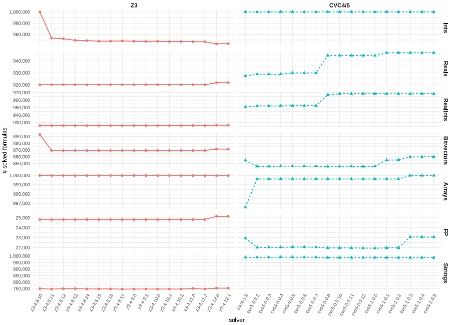
Number of solved formulas (line plots)



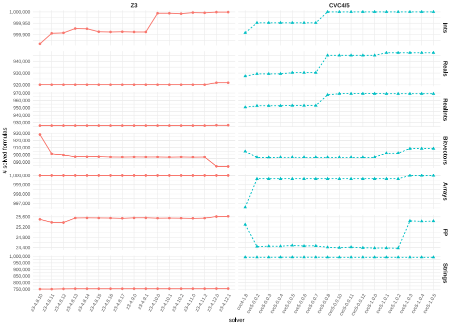
(a) **T=0.03125**



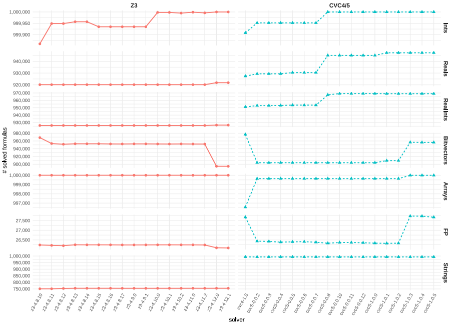
(b) **T=0.0625**



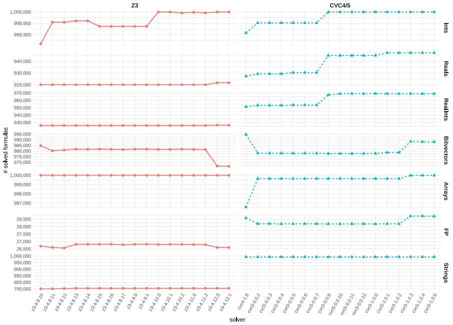
(c) **T=0.125**



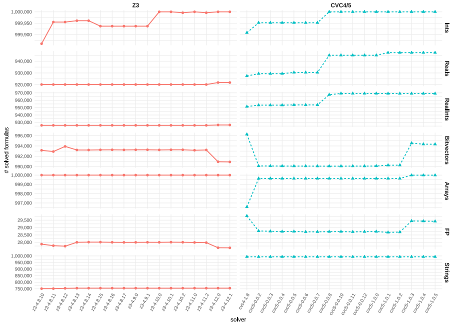
(d) **T=0.25**



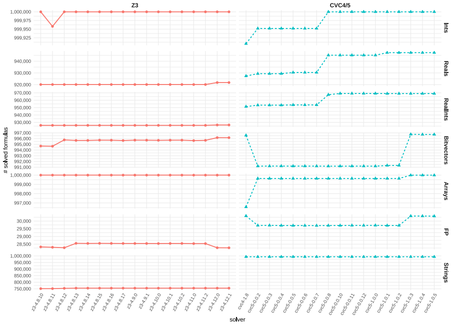
(e) **T=0.5**



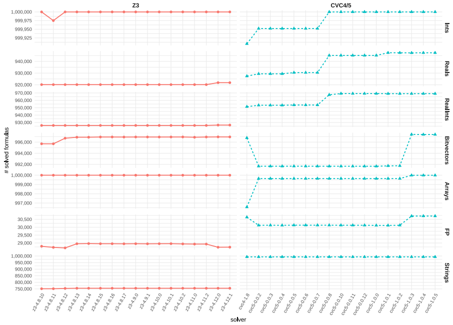
(f) **T=1.0**



(a) **T=2.0**



(b) **T=4.0**



(c) **T=8.0**

CURRICULUM VITAE

PERSONAL DATA

Name	Dominik Winterer
Date of Birth	August 28, 1991
Place of Birth	Villingen-Schwenningen, Germany
Citizen of	Germany

EDUCATION

2019–2024	Ph.D. in Computer Science ETH Zurich Zürich, Switzerland
2015– 2018	MSc. Computer Science Albert-Ludwigs-Universität Freiburg, Germany Freiburg im Breisgau, Germany
2011– 2015	BSc. Computer Science Albert-Ludwigs-Universität Freiburg, Germany Freiburg im Breisgau, Germany