Finding and Understanding Incompleteness Bugs in SMT Solvers

Mauro Bringolf
Department of Computer Science
ETH Zurich, Switzerland
mauro@bringolf.com

Dominik Winterer
Department of Computer Science
ETH Zurich, Switzerland
dominik.winterer@inf.ethz.ch

Zhendong Su
Department of Computer Science
ETH Zurich, Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

We propose Janus, an approach for finding incompleteness bugs in SMT solvers. The key insight is to mutate SMT formulas with local weakening and strengthening rules that preserve the satisfiability of the seed formula. The generated mutants are used to test SMT solvers for incompleteness bugs, *i.e.*, inputs on which SMT solvers unexpectedly return unknown. We realized Janus on top of the SMT solver fuzzing framework YinYang. From June to August 2021, 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 19 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.

ACM Reference Format:
Mauro Bringolf, Dominik Winterer, and Zhendong Su. 2022. Finding and Understanding Incompleteness Bugs in SMT Solvers. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3551349.3560435

1 INTRODUCTION

Satisfiability Modulo Theories (SMT) solvers are fundamental tools for software engineering and programming language advances *e.g.*, symbolic execution [7, 12], program synthesis [23], solver-aided programming [24], and program verification [9, 10]. An SMT solver returns sat on an input formula φ if there is an assignment to φ's variables that evaluates the formula to true, unsat if there is no such assignment and unknown if the SMT solver cannot decide the 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 [7], Microsoft's SAGE [13]), an access policy of a web service (*e.g.* AWS's Zelkova [2]), or a model of a safety-critical system (*e.g.* AdaCores's Spark [1]). 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. Fig. 1 shows a incompleteness bug in Z3 [8] which returns unknown on a simple SMT formula. 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. 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 unsatisfiability. We reported this bug to the issue tracker of Z3. Z3's main developer promptly fixed it.

```
$ z3 bug.smt2
unknown

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

Fig. 1: Command-line trace showing an incompleteness bug in SMT solver Z3 on a simple SMT formula.
https://github.com/Z3Prover/z3/issues/5376

Incompletenesses in SMT solvers. 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

After careful analysis, the developer *"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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560435

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

> *"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

The first comment is from a developer of LLVM static analyzer Alive2 [16], the second from Haskell verifier SBV [11] and the third issue is a query of software verifier SMACK [21]. Incompletenesses are not only a Z3 issue. On CVC5's issue tracker, users report similar experiences [2]. SMT solver developers address such issues by adhoc fixes, pointing to the logic's undecidability and suggesting workarounds. Such hacks can lead to bugs masked as unknowns and suggest that robust SMT solving is impossible.

***Incompletenesses Testing with* `Janus`.** We propose `Janus`, an incompleteness testing tool for SMT solvers. The key idea is to weaken/strengthen seed formulas to generate mutant formulas and stress-test the SMT solvers for incompleteness bugs. We consider two types of incompletenesses in SMT solvers: (1) *regressions incompleteness* and (2) *implication incompleteness*. Regression incompleteness occurs when recent cause the SMT solver to be unexpectedly incomplete. Implication incompleteness occurs when small changes on the input formula, e.g. replacing `=` by `>=`, cause the SMT solver to be incomplete. We believe these notions to be helpful for developers in understanding the incompletenesses of their SMT solvers. We have built `Janus` which can detect both types of incompleteness bugs. From June - August 2021, we have been conducting a fuzzing campaign with `Janus`. We have reported 31 incompleteness bugs to the issue trackers of the state-of-the-art solvers Z3 and CVC5. The developers have appreciated our reports and have modified the solvers based on them: Our reports uncovered functional bugs in algorithms, led to new rewrite rules, changed precedence orders among existing rewrite rules, extended tactics *etc.* `Janus`'s mutation rules are simple by design, for developers to inspect local changes in formulas. We open-sourced `Janus` on Github. [3]

Contributions. We make the following contributions:

- **Incompleteness testing**: We introduce and formalize incompleteness testing for SMT solvers. While unexpected incompletenesses in SMT solvers have been a known issue for quite a long time, to the best of our knowledge, we are the first to formalize and address this problem.
- **Approach & tool**: We propose `Janus`, an approach for finding incompleteness bugs in SMT solvers with weakening and strengthening rules. `Janus` can help SMT solver developers to detect and understand incompleteness bugs.
- **Bug hunting campaign**: With `Janus`, we conduct a 3-month extensive testing campaign for incompletenesses bugs in the two state-of-the-art SMT solvers Z3 and CVC5. We have reported 31 bugs, out of which 26 were confirmed and 19 got fixed by the developers. The developer appreciated our bug reports and commented on our issues with "Thank you for reporting this issue.", "Thank you for the input." *etc.*

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

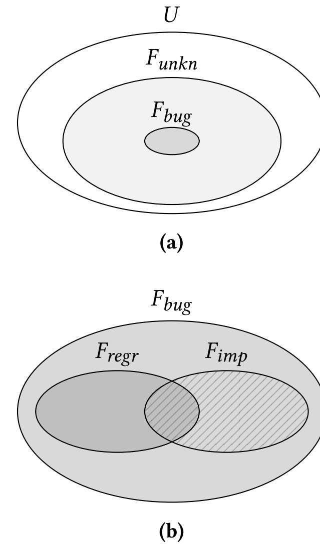

Fig. 2: Classification of incompleteness bugs: Formulas F_{regr} and F_{imp} are our targets for incompleteness testing.

2 PROBLEM STATEMENT

We consider the problem of finding incompleteness bugs F_{bugs} *i.e.*, unexpected incompletenesses in SMT solvers among the set of inputs F_{unkn} on which the SMT solver returns `unknown` from the universe of SMT formulas U. We approximate F_{bugs} this by the following taxonomy with two notions of incompleteness bugs.

Definition 1 (Incompletenesses in SMT solvers). *We distinguish the following two types of incompletenesses:*

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

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

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

Fig. 2 shows a classification of incompleteness bugs: (a) Universe U of SMT formulas handled by an SMT solvers contains formulas for which the SMT solver returns an `unknown` result (F_{unkn}) and their subset the incompleteness bugs (F_{bugs}). (b) F_{bugs} contains regression incompletenesses F_{regr} and implication incompletenesses F_{imp}. Regression incompletenesses are caused by (recent) code changes leading to an incompleteness on previously decided formulas. Typically they affect client software that worked correctly with an older version of the SMT solver but fails after updating the SMT solver. As an example, consider Fig. 3a where Z3's trunk version can not decide the input formula returning `unknown` but legacy version Z3 4.8.10 determines it to be `unsat`. We reported this issue on the issue tracker of Z3. It got fixed within a week by the Z3 developers. Implication incompletenesses occur when an SMT solver can decide a given input formula φ but minor changes in the formula (the mutation to φ') cause the solver to report `unknown`. Such formula pairs can suggest possible improvements for SMT solvers, *e.g.*, to

```
$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)

Fig. 3: The two types of incompletenesses in SMT solvers: (a) Regression incompleteness on a simple formula: legacy Z3 4.8.10 decides this formula as unsat, while the trunk version returned (Z3 #5338). (b) Implication incompleteness in CVC5 caused by minor operator change from = to >= in the formula (CVC5 #7009).

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 Fig. 3b where the operator change from = 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.

3 JANUS

This section presents Janus, our approach to tackle regression and implication incompletenesses. We present (1) an approach overview, (2) introduce necessary background and (3) *Weakening and Strengthening*, the core technique in Janus for incompleteness testing.

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 separate examples.

Finding regression incompletenesses. Fig. 4 shows a mutation chain of Janus for finding regression incompletenesses (from left to right). Janus starts with a seed formula formula on which both Z3 and legacy Z3-4.8.10 return sat (step 1). Janus then chooses a 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 an actual bug that we reported to the Z3's issue tracker.

Finding implication incompletenesses. Fig. 5 shows a mutation chain of Janus for finding implication 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. This results in a mutated formula (step 2) which is 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 (step $n-1$ and 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.

3.2 Background

This section gives background on (1) SMT-LIB, the input language of SMT solvers, (2) logics of SMT-LIB, and (3) basic definitions.

Input language. An SMT-LIB script consists of commands instructing an SMT solver to create formulas and process them. We focus on the following language subset: `declare-const` to declare a constant, `assert` to specify a constraint and `check-sat` to check satisfiability of the asserted formula. Multiple assertions can be viewed as the conjunction of each constraint. We write the universal and existential quantification of a term `t` over a variable `x` with sort `T` as `(forall ((x T)) t)` and `(exists ((x T)) t)` respectively.

Logics. SMT-LIB's logics can be roughly subdivided into quantified (no prefix) and quantifier-free (QF), linear (L) and non-linear (N) theories and the corresponding theory type. For instance, QF_LIA is quantifier-free linear integer arithmetic, NRA is nonlinear quantified real arithmetic, QF_BV is quantifier-free bitvector logic, and A defines quantified array logic. Most logics are NP-hard, some are even undecidable. For a comprehensive list of logics and their complexities, we refer to the SMT-LIB website [3].

Basic definitions. We consider first-order logic formulas of the satisfiability modulo theories (SMT). Such a formula φ is satisfiable if there is at least one assignment (called model) on its variables under which φ evaluates to true. Otherwise, φ is unsatisfiable. Formulas are realized by SMT-LIB programs [3] which we view as abstract syntax trees. We use standard notions of typed higher-order logic, such as term, quantifier, function, *etc.*, and write expression 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. We distinguish between multiple occurrences of syntactically equal parts within a formula. We define $\varphi[F \mapsto G]$ to be the formula represented by an abstract syntax tree of an SMT program where F is replaced by G.

3.3 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 SMT-LIB formulas and then introduce the parity concept for subformulas.

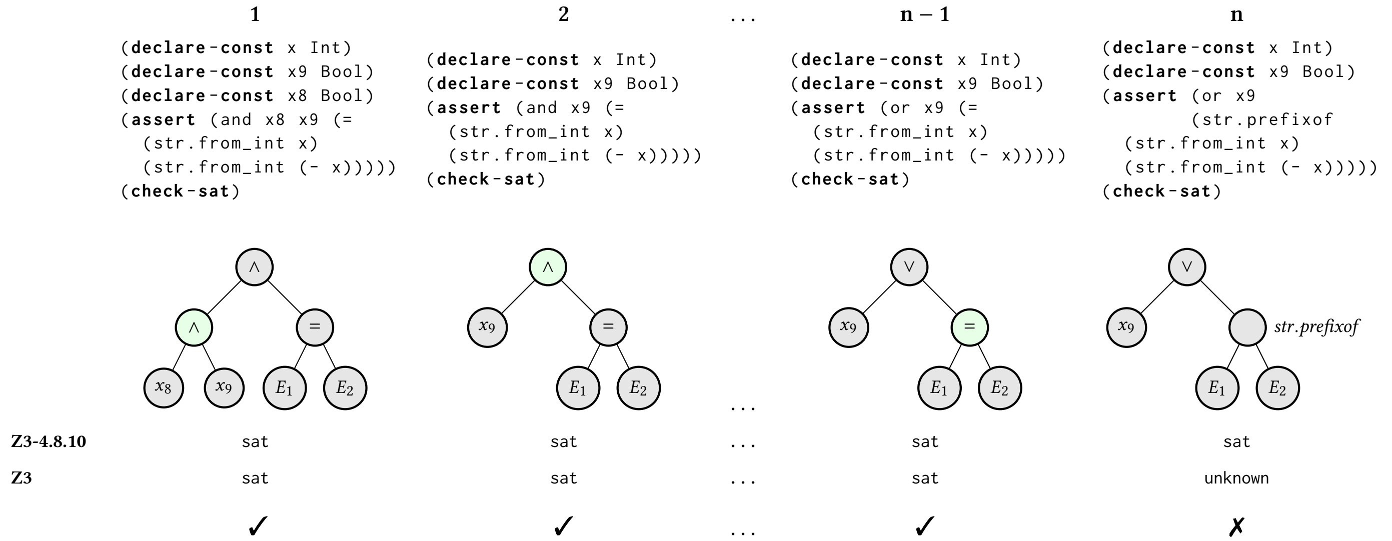

Fig. 4: A sample mutation chain illustrating how Janus finds a regression incompleteness in Z3 (Z3 #5381).

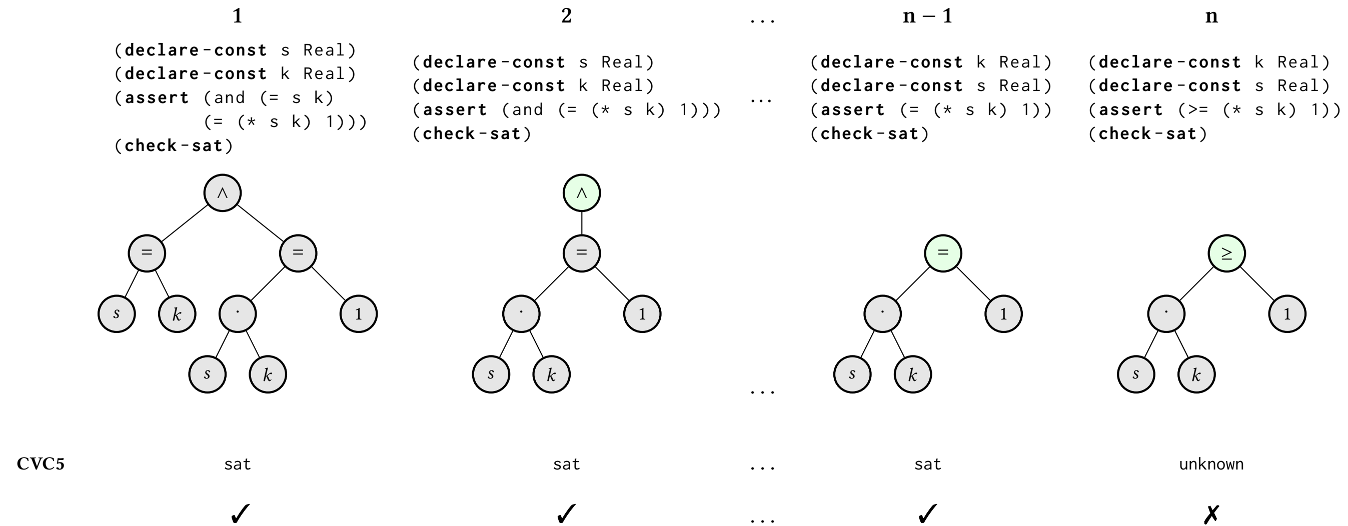

Fig. 5: A sample mutation chain illustrating how Janus finds implication incompleteness in CVC5 (CVC5 #7009).

Definition 2 (weaker/stronger). *Let φ_1, φ_2 be formulas with the same set of free variables* i.e., *$FV(\varphi_1) = FV(\varphi_2) = \{x_1, \cdots, x_n\}$. We call φ_1* weaker *than φ_2 if $\forall x_1, \ldots, x_n : \varphi_2 \rightarrow \varphi_1$.*

Definition 3 (parity). *For a formula φ with a subformula F, we define parity(F, φ) as*

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

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

The parity function 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 1. *Let φ be a formula with a subformula F. For any G weaker than F, we have:*

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

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

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 $\varphi = \varphi_1 \land \varphi_2$: 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:

$$\forall x_1, \ldots, x_n \colon \varphi_1 \land \varphi_2 \to (\varphi_1 \land \varphi_2)[F \mapsto G]$$

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:

$$\forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \to (\exists x_n \colon \varphi_1[F \mapsto G])$$
$$\Longleftrightarrow \forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \to ((\exists x_n \colon \varphi_1)[F \mapsto G])$$

Since $FV(\varphi) = FV(\varphi_1) - \{x\}$, concluding the proof. □

Theorem 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) F positive, φ satisfiable $\Longrightarrow \varphi[F \mapsto F_w]$ satisfiable
(2) F negative, φ satisfiable $\Longrightarrow \varphi[F \mapsto F_s]$ satisfiable
(3) F negative, φ unsatisfiable $\Longrightarrow \varphi[F \mapsto F_w]$ unsatisfiable
(4) F positive, φ unsatisfiable $\Longrightarrow \varphi[F \mapsto F_s]$ unsatisfiable

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

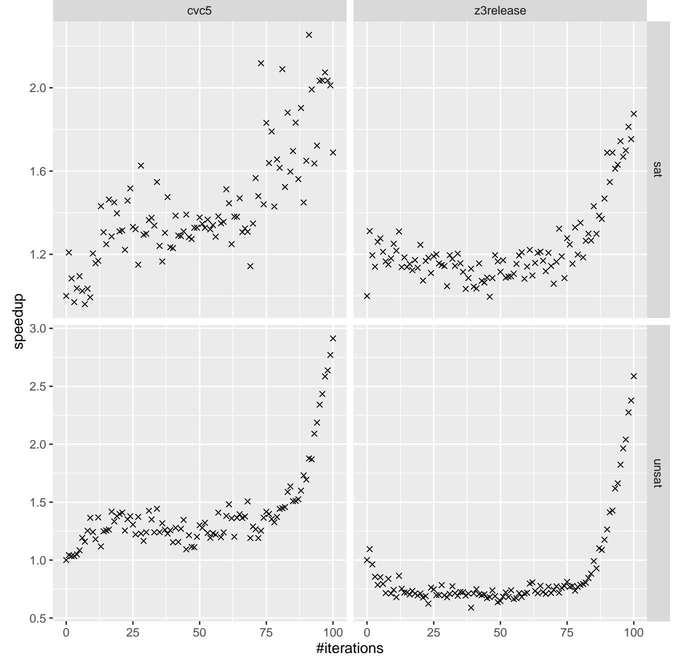

Fig. 6: Z3 and CVC5 runtime performance averages on 1000 evenly distributed sat/unsat nonlinear arithmetic benchmarks per weakening/strengthening iteration (1-100). Timeout: 8 seconds, files on which timeouts occurred excluded.

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' should hence 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 1000 nonlinear benchmarks (500 satisfiable, 500 unsatisfiable) and measured Z3 and CVC5's runtime performance, after 100 weakening and strengthening steps and measured the average speedup. Fig 6 shows the results of this experiment.

Table 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.

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			$\varphi, \varphi_1, \varphi_2$ are boolean formulas
	$\varphi_1 \land \varphi_2$	$\varphi_1 \lor \varphi_2 \mid \varphi_1$	
	$\varphi_1 \oplus \varphi_2$	$\varphi_1 \lor \varphi_2$	$\oplus$ is the logical xor
	$\forall x\colon \varphi \mid \varphi[x \mapsto B]$	$\exists x\colon \varphi$	B is an expression of same type as x
	$x_1 = \ldots = x_n$	$f(x_1) = \ldots = f(x_2)$	$x_1, \cdots, x_n$ are terms of arbitrary type f: SMT-LIB built-in function
	$\varphi_1 \lor \varphi_2$	$\exists b\colon\ ite(b, \varphi_1, \varphi_2)$	*ite* is the if-than-else operator b is a boolean variable
	$ite(B, \varphi_1, \varphi_2)$	$B \rightarrow \varphi_1 \mid \neg B \rightarrow \varphi_2$	
	$\varphi_1 \rightarrow \varphi_2$	$ite(\varphi_1, \varphi_2, \top) \mid ite(\neg\varphi_1, \top, \varphi_2) \mid \forall b\colon (\varphi_1 \land b \rightarrow \varphi_2 \land b)$	
	$\varphi_1 \lor \varphi_2$	$\neg\varphi_1 \rightarrow \varphi_2$	
	$\forall x.\varphi$	$\varphi[x \mapsto B]$	
	φ_1	$\varphi_1 \lor \varphi_2$	
String			s_1, s_2, s_3 are strings
	$s_1 = s_2$	$s_1 \neq (s_1 \mathbin{++} s_3) \mid s_1 \leq_s s_2 \mid prefixof(s_1, s_2) \land suffixof(s_1, s_2) \mid prefixof(s_1, s_2) \land prefixof(s_2, s_1) \mid contains(s_1, s_2) \mid suffixof(s_1, s_2) \land suffixof(s_2, s_1) \mid prefixof(s_1, s_2) \mid suffixof(s_1, s_2)$	++: 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			$r, r_1, \cdots, r_n$ are regexes, $n \in \mathbb{N}$
	r	r^+	+: Kleene plus
	r	$loop(1, n, r)$	$loop(i, n, r) = L(r)^i \cup \cdots \cup L(r)^n$
	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 fixed strings s_1, s_2 choose strings s_3, s_4 s.t. $range(s_1, s_2) range(s_3, s_4)$

Janus' Implementation. We built Januson top of the SMT solver testing framework YinYang [27] in 1.5k lines of Python code. Our implementation first parses and typechecks a given seed formula before incrementally applying randomly selected mutation rules. After a fixed number of mutations, we restart the mutation chain from the seed. Typechecking allows us to apply rules only when they are definitely 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 incompletenesses.

4 EVALUATION

From June 2021 - August 2021, we have been conducting a fuzzing campaign with Janus. We deployed Januson a 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 [8] and CVC5 [4] because (1) they are the most popular SMT solvers (Z3 has $6,7k$ stars on GitHub and CVC5 has 500+ stars on GitHub) (2) Z3 and

Alg. 1: Interestingness test for implication incompletenesses

```
1  Procedure interestingness_implication(φ, m, S):
2      r1 ← S(φ);
3      if r1 ∈ {sat, unsat} then
4          candidates ← {e ∈ expr(φ)| m is applicable to e} ;
5          for c in candidates do
6              ψ ← apply m to c in φ ;
7              r2 ← S(ψ) ;
8              if r2 = unknown then
9                  return 0 ;
10     return 1 ;
```

CVC5 support most logics of the SMT-LIB standard, while the other SMT solvers only partially support the SMT-LIB standard (3) most applications depending on SMT solvers use either Z3 or CVC5 (or even both) since both feature APIs in C++, Python *etc.*. 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. [4]

Bug trigger reduction. We reduce bug triggers of both regression and implication incompletenesses with the SMT-specific test-case reducer ddsmt [18]. 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 respective 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. Alg. 1 shows an algorithm 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 SMT solver 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 that is the case, we return 0 indicating that formula φ is an interesting input.

Bug trigger selection. In our experiments, fuzzing for incompletenesses 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 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. Fig. 7a 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 19 bugs got fixed. Strikingly, while there are 8 fixes in Z3, only one incompleteness bug 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 2 implication incompleteness bugs (see Fig. 7b). 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 bug was rejected by Z3's main developers since fixing it would interfere with axioms that string solvers should create on the fly. Yet another one disappeared after a recent code change in Z3.

Logics distribution among the bugs. Consider Fig. 8 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. In CVC5, many bugs are in quantified string logic while in Z3 many bugs are the in nonlinear logics.

Duration of the campaign & Statistics. Consider Fig. 9. The bug findings are evenly distributed for the fuzzing campaign. 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. Totally, we then reported 31 bugs to the issue trackers of the solvers for the developers to inspect.

5 BUG SAMPLES

This section analyzes exemplary bug reports to illustrate the diverse incompleteness bugs that Janus can find, all depicted in Fig. 10.

Faulty MBQI implementation in Z3 (Fig. 10a). The formula shows a bug in Z3's implementation of Model-Based Quantifier Instantiation (MBQI), a procedure for quantifier elimination. For

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

Status	Z3	CVC5	Total
Reported	13	18	31
Confirmed	8	18	26
Fixed	8	11	19
Rejected	5	0	5

(a)

Type	Z3	CVC5	Total
Regression	7	12	19
Implication	1	6	7

(b)

Fig. 7: (a) Statuses of incompleteness bug reports, (b) types among the confirmed incompleteness bug reports.

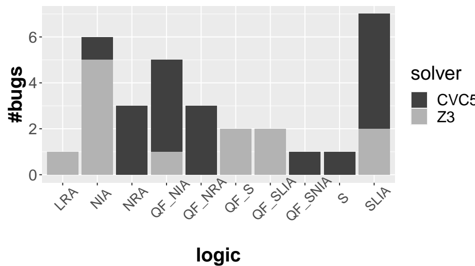

Fig. 8: Bug logic distribution of the reported bugs.

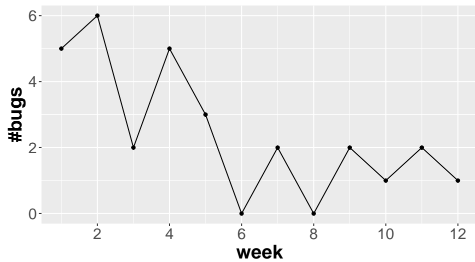

Fig. 9: Incompleteness bugs reported from the start of the bug hunting campaign (week 1) to its end (week 12).

the reported formula, MBQI repeatedly guesses values for universally quantified variables, *i.e.*, x in our case. However, Z3 fails on solving this simple formula, which is unexpected as many values for x would satisfy the formula. Interpreting the semantics of the "Is there is 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 (Fig. 10b). 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. This prevented solving this formula. 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 rewriter precedences.

Completeness regression in Z3's LRA logic (Fig. 10c). The formula belongs to 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 and false). Hence, we would expect SMT solvers to decide the formula. However, Z3 returns `unknown` for it. Z3's main developer fixed the bug by refining the default tactic of Z3.

Incompleteness bug on string formula in Z3 (Fig. 10d). The formula equates two `str.is_digit` expressions, each with a free string variable as single argument. Clearly, 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 implication incompleteness and got promptly fixed.

Completeness regression in CVC5's QF_NRA logic (Fig. 10e). The formula shows a completeness regression in CVC5 on a formula which 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 (Fig. 10f). The formula contains a single existential quantifier and could not be decided 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 the issue to one of his fellow developers for fixing.

Discovering a new `str.replace` rewrite in Z3 (Fig. 10g) This formula contains the term `(str.replace "" va "")` which is equivalent to `""`. Z3 returned `unknown` for the shown formula, but correctly decided 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.

```
1 (declare-const x Int)
2 (assert (forall
3         ((v Int)) (= v (* x x))))
4 (check-sat)
```

(a) Incompleteness bug in Z3 caused by faulty MBQI implementation.
https://github.com/Z3Prover/z3/issues/5376

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

(b) Incompleteness in CVC5 caused by faulty rewrite precedence rules.
https://github.com/cvc5/cvc5/issues/6717

```
1 (declare-const x2 Bool)
2 (declare-const x9 Bool)
3 (declare-fun x () Real)
4 (assert (< x (ite (forall
5 ((x Bool)) (ite x x9 x2)) 0.0 1.0)))
6 (check-sat)
```

(c) Completeness regression in Z3 on formula from a decidable logic (booleans + linear real arithmetic)
https://github.com/Z3Prover/z3/issues/5340

```
1 (declare-const P String)
2 (declare-const T String)
3 (assert (=
4 (str.is_digit T)
5 (str.is_digit P)))
6 (check-sat)
```

(d) Incompleteness bug on string formula in Z3 caused by unhandled `str.is_digit`.
https://github.com/Z3Prover/z3/issues/5491

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

(e) Completeness regression in CVC5's QF_NRA logic.
https://github.com/cvc5/cvc5/issues/6798

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

(f) Regression with existential quantifier elimination.
https://github.com/cvc5/cvc5/issues/6727

```
1 (declare-const x Int)
2 (declare-fun T () Int)
3 (declare-fun va () String)
4 (assert (distinct (str.from_int T)
5 (str.replace va (str.replace ""
6 va "") (str.from_int (- x)))))
7 (check-sat)
```

(g) Regression bug in Z3 which led to new `str.replace` rewrite being implemented.
https://github.com/Z3Prover/z3/issues/5399

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

(h) Order sensitivity of `or` in CVC5 which raised a discussions among the developers.
https://github.com/cvc5/cvc5-projects/issues/279

Fig. 10: Selected bug samples in Z3 and CVC5.

Order sensitivity of `or` in CVC5 (Fig. 10h). 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 apparently 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 decided that this incompleteness is an acceptable artifact of the solver's internals. Nevertheless, they kept the issue in a collection of challenges.

6 RELATED WORK

Our approach is particularly related to the prior works on SMT solver robustness testing [6, 17, 19, 25, 26]. One closely related work is Bugariu and Müller's approach [6]. Bugariu and Müller's formula 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 [28]. 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. Blotsky *et al.* [5] proposed StringFuzz focusing on performance issues in string logic. Besides domain-specific approaches, our approach is related to performance fuzzing. SlowFuzz [20] is an approach to finding complexity vulnerabilities in sorting and compression algorithms, PerfFuzz [15] is a tool using coverage guidance to find performance bugs along frequently executed program paths, and Ga-Proof [22] 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 CONCLUSION

We have introduced Janus, an effective testing tool to uncover incompleteness bugs in SMT solvers. The key idea is to feed strengthened/weakened SMT formulas to the SMT solvers under test to uncover two types of incompleteness bugs – regressions and implication incompletenesses. We realized Janus on top of the SMT solver fuzzing tool yinyang. Between June and August 2021 we have been stress-testing Z3 and CVC5, the state-of-the-art SMT solvers. We reported 31 incompleteness bugs of which 26 were confirmed and 19 fixed by solver developers. Most reports uncovered functional bugs in the SMT solvers that have not manifested as soundness or crash bugs. This is the first work introducing and exclusively targeting incompleteness bugs in SMT solvers. As future work, we plan to research ways of detecting incompleteness bugs beyond regressions and implication incompletenesses. Reducing and selecting incompleteness bugs for reporting is a slow and manual process. Hence, we also plan to automate this process.

ACKNOWLEDGMENTS

We thank the anonymous ASE reviewers for their valuable feedback. Our special thanks go to the CVC5 and Z3 developers, especially Andrew Reynolds, Gereon Kremer, Haniel Barbosa, Nikolaj Bjørner *etc.*, for useful information and addressing our bug reports. This work was partially supported by an Amazon Research Award.

REFERENCES

[1] AdaCore. 2022. SPARK. Retrieved 2022-09-07 from https://github.com/AdaCore/spark2014

[2] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *FMCAD '18*. 1–9.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2019. The Satisfiability Modulo Theories Library (SMT-LIB). Retrieved 2022-08-09 from www.SMT-LIB.org

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *SMT '10*.

[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV '18*. 45–51.

[6] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE '20*. 1459–1470.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI '08*. 209–224.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08*. 337–340.

[9] Rob DeLine and Rustan Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs.* Technical Report.

[10] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *JACM* (2005), 365–473.

[11] Levent Erkok. 2022. SBV: SMT Based Verification in Haskell. Retrieved 2022-09-07 from https://github.com/LeventErkok/sbv

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI '05*. 213–223.

[13] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Queue* (2012), 20–27.

[14] Andrei Lascu, Alastair F. Donaldson, Tobias Grosser, and Torsten Hoefler. 2022. Metamorphic Fuzzing of C++ Libraries. In *ICST '22*. 35–46.

[15] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *ISSTA '18*. 254–265.

[16] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. *PLDI '15*, 22–32.

[17] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *FSE '20*. 701–712.

[18] Aina Niemetz and Armin Biere. 2013. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *SMT '13*. 36–45.

[19] Aina Niemetz, Mathias Preiner, and Clark Barrett. 2022. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *CAV '22*. 92–106.

[20] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *CCS '17*. 2155–2168.

[21] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *CAV '14*. 106–113.

[22] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2015. Automating Performance Bottleneck Detection Using Search-Based Application Profiling. In *ISSTA '15*. 270–281.

[23] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph. D. Dissertation. University of California at Berkeley.

[24] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI '14*. 530–541.

[25] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusal Effectiveness of Type-Aware Operator Mutation. *OOPSLA '20*, 1–25.

[26] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. *PLDI '20*, 718–730.

[27] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2022. yinyang: a fuzzer for SMT solvers. Retrieved 2022-09-07 from https://github.com/testsmt/yinyang

[28] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal Approximation Enumeration for SMT Solver Testing. In *FSE '21*. 1141–1153.