Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration

DOMINIK WINTERER, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

We introduce ET, a grammar-based enumerator for validating SMT solver correctness and performance. By compiling grammars of the SMT theories to algebraic datatypes, ET leverages the functional enumerator FEAT. 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 84 were confirmed and 40 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. Due to its systematic testing and efficiency, we further advocate ET's use for continuous integration.

CCS Concepts: • **Software and its engineering → Formal methods**.

Additional Key Words and Phrases: SMT solvers, Fuzz testing, Grammar-based enumeration

ACM Reference Format:
Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 355 (October 2024), 24 pages. https://doi.org/10.1145/3689795

1 Introduction

Satisfiability modulo theory (SMT) solvers are foundational for many applications and systems in academia [14, 18, 21, 37, 41] and industry [1, 3, 22]. 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 [27, 42, 43, 46]. Z3 [17] and cvc5 [4] 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 Fig. 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.

Authors' Contact Information: Dominik Winterer, ETH Zurich, Zurich, Switzerland, dominik.winterer@inf.ethz.ch; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

ACM 2475-1421/2024/10-ART355
https://doi.org/10.1145/3689795

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

Fig. 1. Critical soundness bug in cvc5 found by 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 [19], 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 [2, 26], (2) because of the small-sized bug triggers it is particularly suitable for identifying performance issues, and (3) it provides bounded correctness guarantees *w.r.t.* the grammar G and the differential oracle. Our empirical evaluation shows that ET is highly effective at finding bugs 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 pipelines of SMT solvers.

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 84 bugs were confirmed and 40 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 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 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 times. Our correctness results reveal that both solvers have greatly matured (see Fig. 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 with a significant decrease from z3-4.8.10 to z3-4.8.11 (see Fig. 2 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

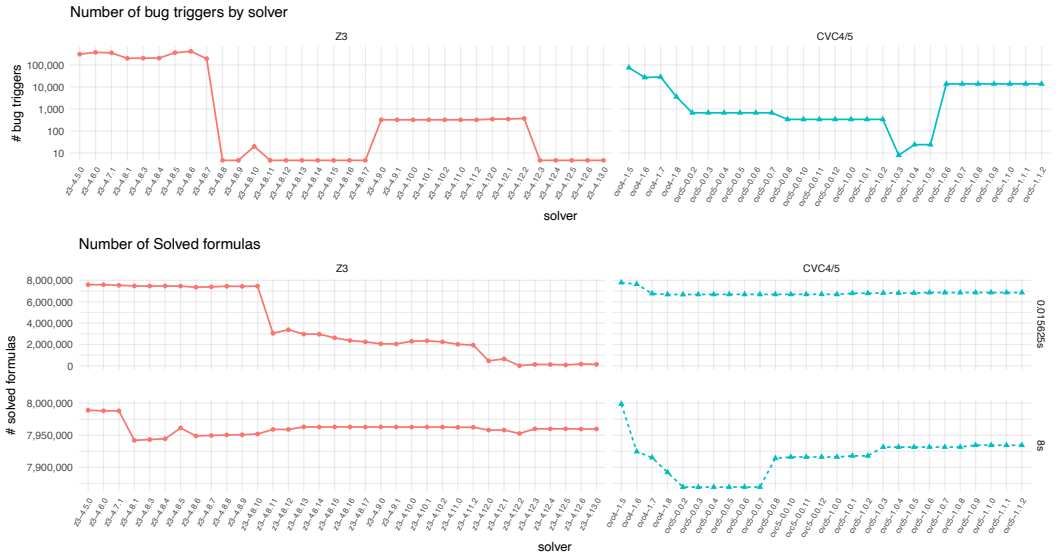

Fig. 2. Evolution results for Z3 & CVC4/cvc5 releases from the last six years. Top: correctness in number of bug triggers. Bottom: performance in number of solved formulas on timeouts (T=0.015625s) and (T=8s).

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 continuing 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% of the bugs.

Main contributions. We make the following contributions:

- We engineered ET, a grammar-based enumerator for validating SMT solver's correctness and performance. By compiling context-free grammars to algebraic datatypes, ET leverages the functional enumeration library FEAT;
- We devised eight grammars for all official SMT theories and utilizing ET with these grammars, we conducted an extensive testing campaign of the SMT solvers Z3 and cvc5. Despite their extensive and continuous testing, we found 102 bugs, 84 were confirmed and 40 were fixed;
- Enabled by ET's systematic testing, we could study the evolution of correctness and performance in Z3 and CVC4/cvc5 releases from the last six years (61 versions) with a total of 8 million tests, and 488 million solver calls; and
- We explored the practicability of ET for CI/CD pipelines on commodity hardware.

Organization of the paper. The rest of the paper is structured as follows. Section 2 illustrates ET via an example. Section 3 provides formal background on our approach and describes the

```
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 PO Store Var_a Bv_term Bv_term PC
```

(b) Compilation of a single production of G_{Arrays}

```
(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) Formula 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}

Fig. 3. Grammar-based enumeration with ET illustrated.

implementation of ET. Section 4 presents our evaluation. In Section 5, we discuss extensibility scalability, and limitations. Section 6 surveys related work, and Section 7 concludes the paper.

2 Illustrative Example

This section gives a brief introduction to SMT and the SMT-LIB language [7] and illustrates grammar-based enumeration of SMT solvers with ET.

SMT and SMT-LIB language. SMT is a collection of logic theories relevant to programming languages including booleans, integers, reals, bitvectors, arrays, floating points, and strings. Given an SMT formula φ, an SMT solver is a tool for solving φ automatically. It returns `sat` if there is an assignment for φ's variables that evaluates the formula to true, `unsat` if there is no such assignment, and `unknown` if it cannot determine φ's satisfiability. SMT-LIB [6] is the standard input language for SMT solvers. We focus on the following language subset of SMT-LIB: `"declare-const"` to declare a variable, `"assert"` to specify a constraint and `"check-sat"` to query the satisfiability.

Grammar-based enumeration in steps. 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 Fig. 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}. As an example, consider the productions of nonterminal `arr_term` (see Fig. 3b) which is compiled into an algebraic datatype of a functional programming language.
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 can then enumerate the N-smallest formulas from G_{Arrays}. Thanks to FEAT's efficiency, the enumeration takes less than 1.5 minutes.
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 (Fig. 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 (Fig. 3d). We have reported it to cvc5's issue tracker and it was fixed recently.

3 Approach

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.

Basic definitions. We consider formulas of the satisfiability modulo theories (SMT). Such a formula φ is satisfiable if there is at least one assignment on its variables under which φ evaluates to true. Otherwise, φ is unsatisfiable. Formulas are represented by SMT-LIB programs [6]. 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.

Having a basic background, we introduce regular tree grammars, show our compilation from context-free to regular tree grammar, and finally show how we realize grammar-based enumeration.

Regular tree grammar. 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 $C(.)$ 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 an RTG.

Definition 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* $S' = S$
- $\Sigma' = \Sigma \cup \{ C_i^{lhs} \mid \ p_i^{lhs} \in P_N \}$
- $P' = \{ lhs \rightarrow C_i^{lhs} rhs_i \mid p_i^{lhs} \in P_N \} \cup P_\Sigma$

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. The next paragraph will present an example on this compilation.

Functional enumeration of algebraic types [Duregard et al. 2012]. We consider the size-based enumerator FEAT as a function $FEAT : \mathbb{N} \rightarrow L(G_{RTG})$ from the natural numbers to the (countable) language of an RTG 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

```
1:  procedure ET(G, O, N)                          ▷ CFG G, oracle O, #tests N
2:      bug_triggers ← []
3:      G_RTG ← RTG(G)                             ▷ Compilation based on Definition 1
4:      for φ_i ∈ FEAT(G_RTG)[0 : N] do            ▷ Loop in parallel
5:          bug_found, trace_i ← O(φ_i)            ▷ Oracle check
6:          if bug_found then
7:              bug_triggers ← bug_triggers.append((φ_i, trace_i))
8:          end if
9:      end for
10: end procedure
```

Fig. 4. Pseudocode of ET's main prcoess.

from the Core theory (see Fig. 5a).

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

Four nonterminals are necessary to realize this formula. Hence changing the `false` to `true`, `a`, or `b` will not increase size. However, the following formula has size 5:

```
(declare-const a Bool) (declare-const b Bool) (assert (not false)) (check-sat)
```

FEAT realizes a partitioning and ordering of $L(G_{RTG})$ based on size. This avoids explicit generation. As a result, FEAT can index each of the elements realizing random access to elements in $L(G_{RTG})$. We write $FEAT(G_{RTG})[0 : N]$ to denote the list of the N-smallest elements in G_{RTG}.

ET's realization. We next describe ET's realization. ET takes as inputs a context-free grammar G, an oracle O, and a desired number of tests N. The following pseudocode shows ET's main process (Fig. 4). 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 a total of 103 lines of Python and Bash script code.

Deriving grammars for the SMT theories. A key ingredient of our approach are the grammars. As a template, we use a generic SMT-LIB grammar from the ANTLR grammar repository. [1] Starting with a generic grammar for SMT-LIB, we pruned all productions unrelated to declarations, assertions, expressions, and a solver query. The resulting grammar describes the most common format of SMT formulas. The resulting grammar describes the most common format of SMT formulas. As a next step, we derived a separate grammar for each SMT theory by adding theory-specific operators along with their typed signatures. We chose two variables and two constants to realize interesting SMT formulas. We derive one grammar per official SMT theory including the booleans (Core), integers (Ints), real numbers (Reals), mixed reals, integers (RealInts), bitvectors (FixedSizeBitVectors), arrays (ArraysEx), floating point numbers (FP), and unicode strings (Strings).

[1] https://github.com/antlr/grammars-v4/blob/master/smtlibv2/SMTLIBv2.g4

We studied the theory specifications to ensure that the grammars covered all operators from the respective theories. [2] 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 Fig. 5 + 6). All grammars are designed such that the resulting SMT-LIB scripts are well-typed. Richer grammars of SMT theories can be devised by modifying existing or creating new grammars.

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 solver releases over the last six years. We finally explore ET's potential as a monitoring tool for continuous integration. ET is available on Github.[3]

Result summary.

- *Many bugs in Z3 and cvc5*: We found 102 bugs, 53 correctness and 49 performance bugs. Among these 84 were confirmed, and 40 were fixed by the developers.
- *Insightful evolution results:* We observe significantly increased reliability 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.

4.1 Research Questions

We aim to answer the following four consecutive research questions:

RQ1 *How effective is ET at bug finding?*
RQ2 *Can we use ET to quantify the reliability 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.*

4.2 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 the research questions RQ2-RQ4, we repeated the experiments three times and averaged the results.

[2] https://smtlib.cs.uiowa.edu/theories.shtml
[3] https://github.com/wintered/ET

```
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

Fig. 5. 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

Fig. 6. Derived grammars for the SMT theories (ctd).

Oracles. We use the following two oracles for our evaluation with ET:

$O_{\textbf{test}}$ 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 60 seconds on all solver calls.[4]

$O_{\textbf{evol}}$ 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 to maximize the chances of catching soundness bugs. Oracle O_{test} is used in RQ1 and oracle O_{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 to test the solvers with larger formulas.

[4]Default modes mean no further options enabled, *i.e.*, using the defaults for random seeds *etc.*

Status	Z3	cvc5	Total
Reported	38	64	102
Confirmed	21	63	84
Fixed	13	27	40
Duplicate	0	2	2
Won't fix	4	1	5

(a)

Type	Z3	cvc5	Total
Soundness	2	11	13
Crash	1	19	20
Invalid Model	10	10	20
Performance	8	23	31

(b)

#Options	Z3	cvc5	Total
default	9	36	45
1	6	27	33
2	6	0	6

(c)

Fig. 7. (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.

Bug types. Let S be an SMT solver under test, S_{ref} a reference solver and φ a formula. We distinguish between the following types of bugs:

- *Soundness bug*: Formula φ triggers a soundness bug in S if $S(\varphi) \neq S_{ref}(\varphi)$.
- *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.
- *Performance bug*: Formula φ triggers a performance bug if the solver does not solve it within a given timeout.

Soundness and invalid model bugs relate to issues with potentially severe consequences on downstream applications. Crash bugs can have diverse causes and are usually considered less severe. Soundness bugs are detected by differential testing. Invalid model and crash bugs are detected by non-zero exit code and matching patterns on standard output and error. To differentiate from performance bugs, we refer to soundness, invalid model, and crashes as correctness bugs. To catch performance bugs, we examine the smallest triggering formula for which a timeout occurs.

Bug triggers. Dozens of bug triggers usually point to the same underlying bug. To avoid duplicate bug reports, we de-duplicated the bug triggers after each fuzzing run with ET 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 determine the formula with the smallest index and report it. 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. Because of the small size of ET's bug triggers, no bug reduction was necessary.

RQ1: How Effective is ET at Bug Finding?

Using ET with oracle O_{test}, we extensively stress-tested the SMT solvers Z3 and cvc5. *We reported* 102 *bugs, out of which* 84 *were confirmed, and* 40 *were fixed* (see Fig. 7a). The bug types are fairly evenly distributed: 13 are soundness bugs, 20 are invalid model, 20 are crashes, and 31 are performance bugs (see Fig. 7b). Of the confirmed bugs, most bugs affect the solver's default modes (45 out of 84), followed by one-option configurations (33 out of 84) and six bugs affect two option configurations (see Fig. 7c).[5] Among the bugs that we reported, there are 53 correctness and 49 performance bugs. We also inspect the theory distribution which we analyze for correctness bugs and performance

[5]Five out of six of these bugs are related to Z3's new core `tactic.default_tactic=smt sat.euf=true`.

bugs separately. 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 earlier findings by cvc5's internal fuzzer Murxla [29], 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 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 [35] 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. 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 (Fig. 8a). 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 (Fig. 8b). The formula realizes the inequality `-a > (1 mod -1) = 0`. Clearly, a negative `a` would satisfy the inequality, hence the formula is satisfiable. However, Z3's new core reports `unsat` for this formula. We reported the bug and it was promptly fixed.

Soundness bug in cvc5's string theory (Fig. 8c). 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 (Fig. 8d). 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 years 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 one of the most sizable that ET found. Almost all other bugs were smaller.

Performance bug in Z3's Ints theory (Fig. 8e). The formula triggers a performance bug in Z3. The formula has a single integer variable `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 (Fig. 8f). 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.

```
(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/5842

```
(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 both Z3 and cvc5:* timeouts on simple bitvector formula.

https://github.com/Z3Prover/z3/issues/6800
https://github.com/cvc5/cvc5/issues/9874

Fig. 8. Selected correctness and performance bugs found by ET in Z3 and cvc5.

Performance bug in cvc5 string (Fig. 8g). 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.

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		

Fig. 9. 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.

Performance bug in both Z3 and cvc5 (Fig. 8h). 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 of the solvers.*

RQ2: Can we Use ET to Quantify the Reliability 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 reliability of SMT solvers. To approach this question, we ran ET with oracle O_{evol} on all consecutive releases of Z3 and CVC4/cvc5 from the last six years (see Fig. 9). 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 each solver call, we chose a timeout of 8 seconds and a memory limit of 1GB.

Number of bug triggers per solver and theory. We present the results in a line plot (Fig. 10) 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 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. Fig. 11). The 8 million formulas break down into 6,355,636 satisfiable and 1,644,364 unsatisfiable. We did not observe any striking patterns.

Z3. Considering the correctness results of Z3 (Fig. 10 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

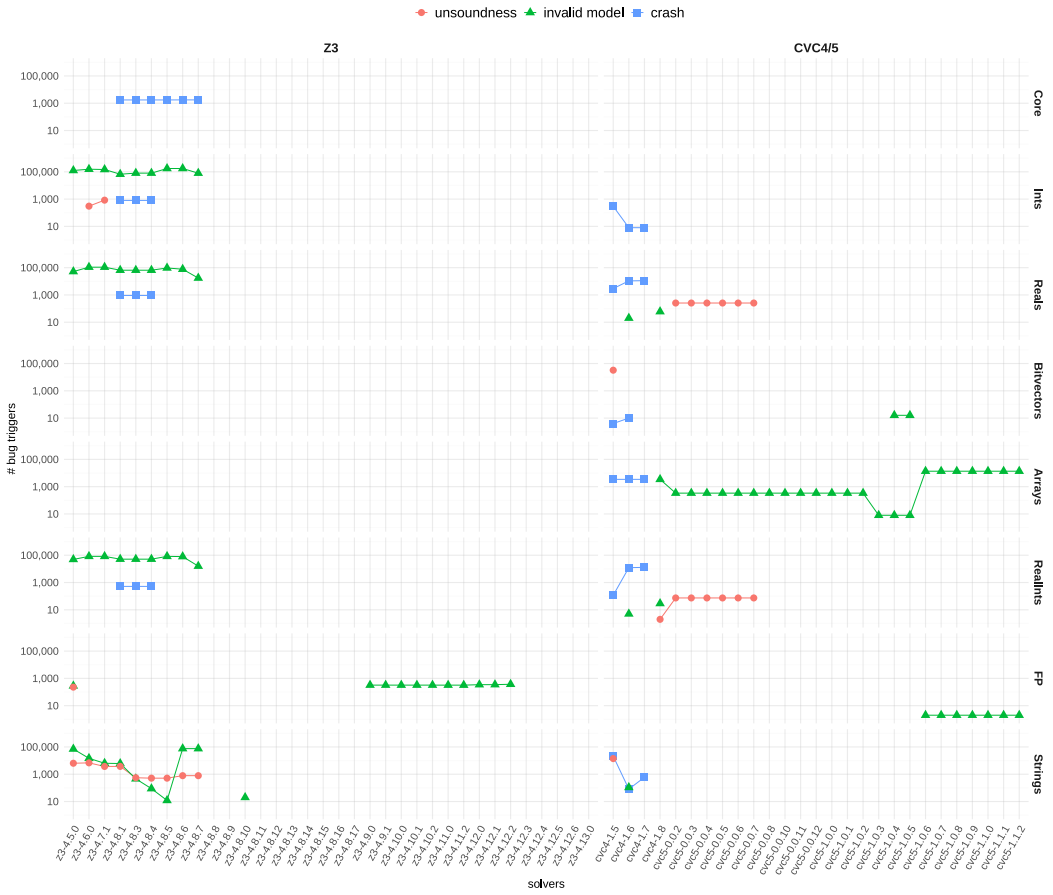

Fig. 10. Bug triggers per theory in all releases of Z3 (left) and CVC4/cvc5 (right) since November 2016.

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.

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 reliability 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	370.34	0.0
z3-4.12.1	0.0	347.0	0.0
z3-4.12.0	0.0	345.34	0.0
z3-4.11.2	0.0	319.34	0.0
z3-4.11.0	0.0	319.67	0.0
z3-4.10.2	0.0	320.67	0.0
z3-4.10.1	0.0	321.0	0.0
z3-4.10.0	0.0	320.34	0.0
z3-4.9.1	0.0	321.0	0.0
z3-4.9.0	0.0	321.0	0.0
z3-4.8.11-17	0.0	0.0	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	187858.34	1759.0
z3-4.8.6	785.0	406640.0	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	360773.34	0.0
z3-4.5.0	6619.0	296769.0	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	46487.67	0.0	27898.0

Fig. 11. Bug triggers in Z3 (left) and CVC4/cvc5 releases (right). Bold: solvers without bugs.

RQ3: Can we Use ET to Quantify The Performance of SMT Solvers?

After noting the improved correctness of the solvers, we next investigate 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 [6] 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 Fig. 12). We have a column for each solver 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 Fig. 14). For a more complete set of plots, we refer to the supplementary material.

Lowest timeout T=0.015625s. Considering the results for Z3 (Fig. 12 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, almost all the rejected formulas were turned into unknowns. Considering the results for CVC4/cvc5 (Fig. 12 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

[6] By jointly solved formulas, we mean all formulas from the set $\{\varphi \mid S(\varphi) \in \{sat, unsat\}$ for all $S\}$.

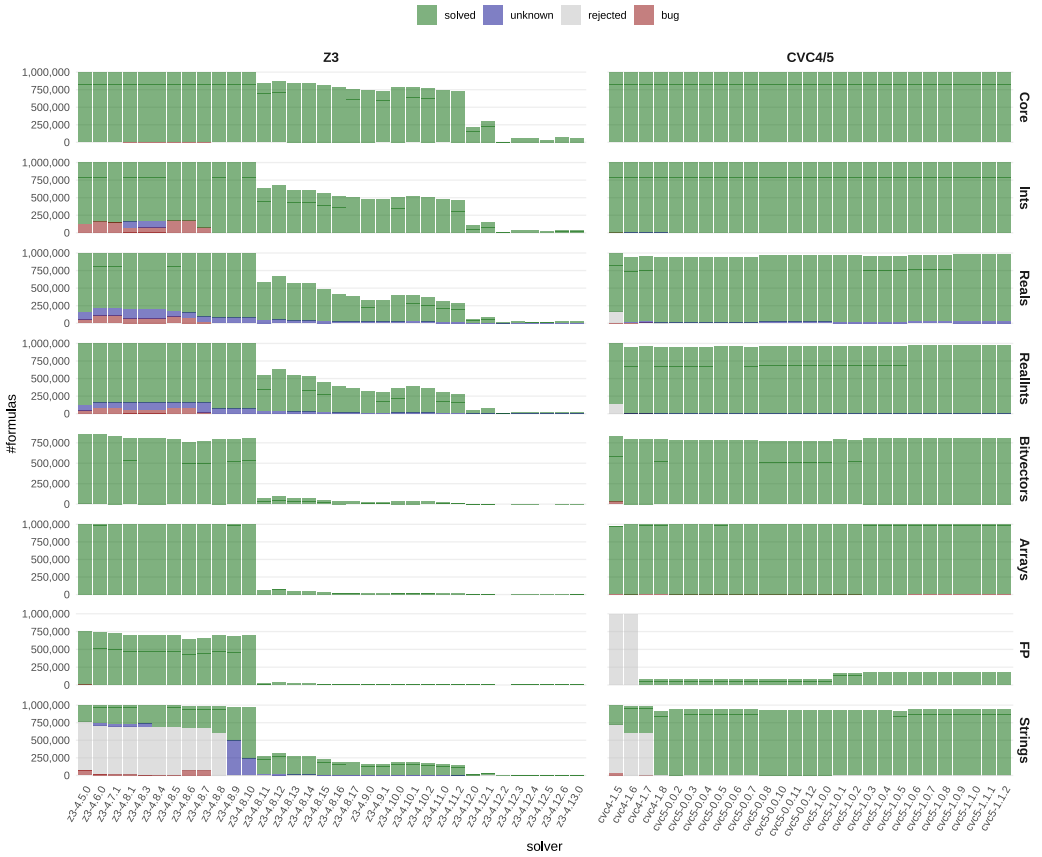

Fig. 12. Number of formulas for lowest timeout T = 0.015625s. White spaces indicate unsolved formulas.

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:

```
989  public:
990 +    ast_table() : chashtable({}, {}, 512 * 1024, 8 * 1024) {}
991      void push_erase(ast * n);
992      ast* pop_erase();
993  };
```

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

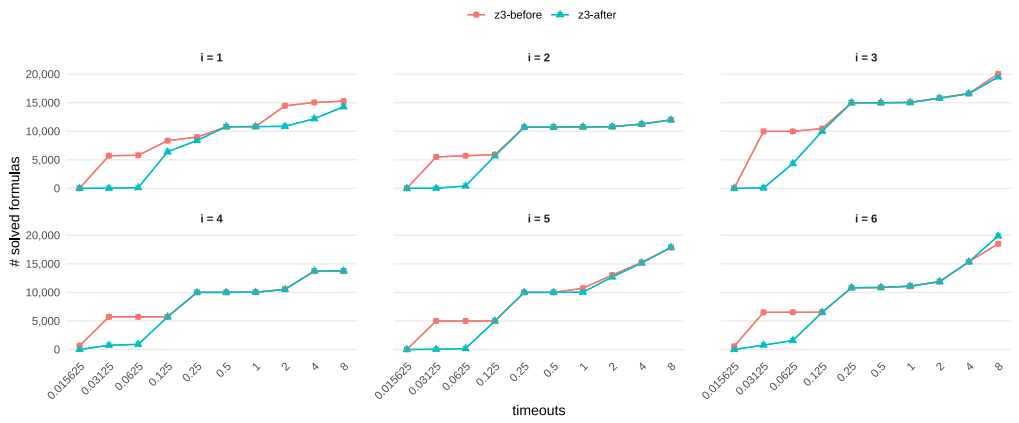

Fig. 13. Results on larger formulas (4 KB and larger) to understand the significant performance decline in Z3: before (z3-before) and after increasing the hash table size (z3-after).

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 $FEAT(idx)$
b) $idx := idx \cdot offset$

where the *offset* is 10^{10} and *idx* is initially set to *start_idx* which varies from 10^{275} to $10^{1,700}$ depending on the grammar. Considering the results (Fig. 13), 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 *i.e.*, it usually vanishes 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 Fig. 14 left), we observe that Z3 solves a constant number of formulas since z3-4.8.8 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 Fig. 15). 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. 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.

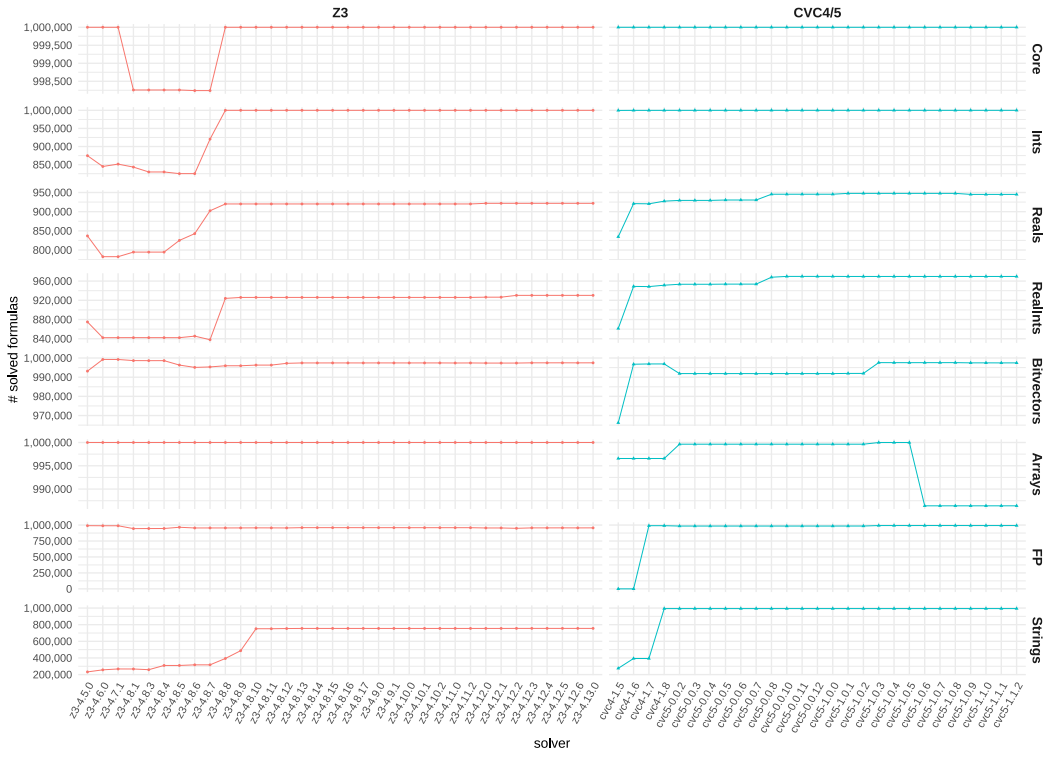

Fig. 14. Number of solved formulas for the highest timeout T=8s.

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 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 bugs trigger (from RQ2) 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% of the bugs, and one hour and 18 minutes for cvc5 to cover 99% of the bugs.

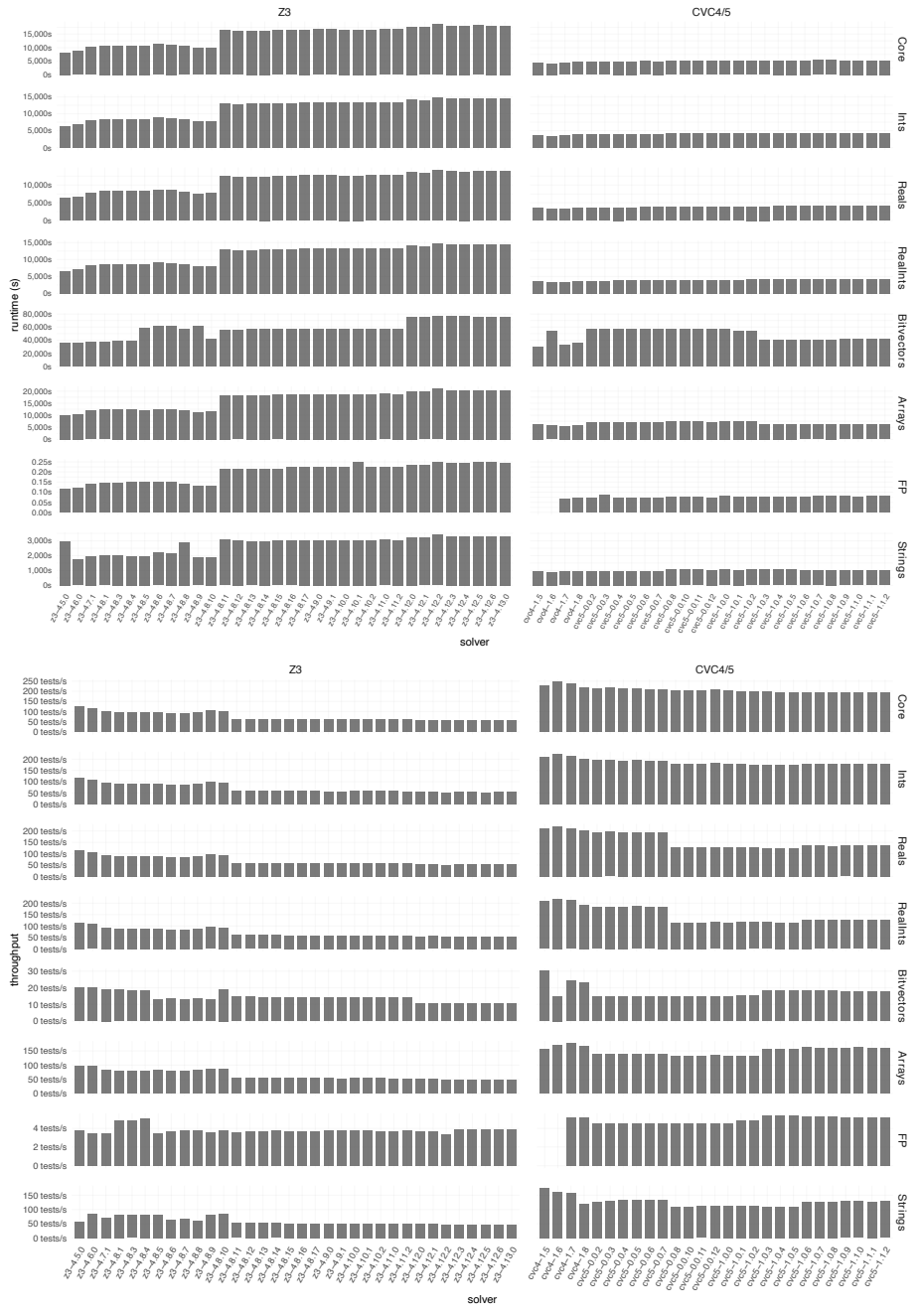

Fig. 15. Top: cumulative runtime on jointly solved formulas. Bottom: Throughput in formulas per second.

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.*

5 Discussion & Threats to Validity

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 outline the threats to validity.

5.1 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, 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, to challenge the resulting combinatorial blow-up, uniform random sampling would need to be added. On a higher level, this shows the complementary strengths and weaknesses of enumerative versus random testing. Pure enumerative testing (without sampling), such as ET, is systematic and yields correctness assurances but needs tailored grammars to be effective. Grammar-based random testing, on the other hand, does not yield correctness assurances but also does not need tailored grammars to be effective.

5.2 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.

5.3 Threats to Validity

We acknowledge the following threats of validity. Since ET exhaustively enumerates tests, we needed to restrict variable and constant count to cover interesting input space in a reasonable number of tests (1 million). In RQ2+3, we hence chose two variables and constants, respectively. Consequently, RQ2 is an underestimate. To understand the extent of this trade-off, we re-ran the experiments of RQ2+3 with 5, 8, and 10 variables. Our observations show progressively fewer bug triggers. This is in line with the small scope hypothesis but also shows another limitation of enumeration and FEAT: With an increasing number of variables, we observe many α-equivalent formulas. 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 three times in an isolated setup, and disabled hyperthreading and frequency scaling.

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 [2009b] 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 [30], a testing tool for Boolector [10], 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 [9, 12]. However, later STORM [27] and YinYang [43] found dozens of soundness bugs in Z3. Even later, OpFuzz, TypeFuzz and Falcon [31, 42, 45, 46] found several hundred bugs in Z3 and CVC4/cvc5. Besides the correctness fuzzers, StringFuzz [9] 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 [36] 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 [19], which belongs to the family of property-based testers. Another similar tool is LeanCheck [28] supporting richer properties than FEAT. ET is loosely related to the enumerative tester SmallCheck [34], and the random property-based tester QuickCheck [15]. In the software engineering community, researchers proposed Bounded Exhaustive Testing, through an approach for testing Galileo, a dynamic fault tree analysis tool [39]. 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 [25] for input generation. Conceptually related to FEAT are two works from the programming language community on declarative rewriting [32] and bounded model checking of Rust typing rules [33] employing a map from the input space to integer indices. More loosely related is skeletal program enumeration (SPE) [47], 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 [13, 23, 24, 44] 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 [38], 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 [8], 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 [16, 20, 40]. Most closely related is the SMT-COMP [5], 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 benchmark set consists of applications that are intentionally challenging for SMT solvers. As a key difference to 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.

7 Conclusion

We changed the perspective on testing SMT solvers from being random and unsystematic to enumerative and systematic. We introduced ET, a highly effective grammar-based enumerator for validating SMT solver correctness and performance. Even though Z3 and cvc5 have been extensively and continuously tested, ET was still able to find 102 bugs while the benefits of other bug-hunting tools have saturated. Out of these bugs, 84 bugs were confirmed and 40 bugs were fixed. As a key advantage, ET's bug triggers are small, making them much suited for identifying performance issues as compared to existing random testing approaches with large formulas. Besides finding bugs, ET's systematic nature allows for a large-scale study on all Z3 and CVC4/cvc5 releases from the last six years (61 solvers) in a total of 8 million tests, and 488 million solver calls. Our results reveal significant improvements in the correctness of both Z3 and CVC4/cvc5, however a performance decline for Z3 on small timeouts. Most notably, both Z3 and CVC4/cvc5 have significantly improved correctness in the theory of Strings, which was previously considered unstable. To the best of our knowledge, our work is the first systematic approach to SMT solver testing. Thanks to ET's efficiency, we believe it to be an ideal monitoring tool for SMT solvers' correctness and performance. In the the future, we will open-source ET, and integrate it into CI/CD pipelines of the SMT solvers. We plan to post regular updates on solver correctness and performance. Encouraged by its effectiveness for SMT solvers, ET yields a fresh, perspective on systematic testing. We aim to adapt ET for testing JavaScript engines, SAT solvers, Golang and other compilers.

Acknowledgments

We thank the anonymous OOPSLA reviewers for their valuable feedback. Our special thanks go to the cvc5 and Z3 developers, especially Andrew Reynolds, and Nikolaj Bjørner for useful information and addressing our bug reports. This work was partially supported by an Amazon Research Award.

References

[1] AdaCore. 2021. SPARK. Retrieved 2023-10-23 from https://github.com/AdaCore/spark2014
[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the "Small Scope Hypothesis". In *POPL '03*. 1–12.
[3] 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.
[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV '11*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14
[5] Clark Barrett, Leonardo de Moura, and Aaron Stump. 2005. SMT-COMP: Satisfiability Modulo Theories Competition. In *CAV '05*. 20–23.
[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2019. The Satisfiability Modulo Theories Library (SMT-LIB). Retrieved 2023-10-2023 from www.SMT-LIB.org
[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *SMT '10*.
[8] 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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06*. 169–190. https://doi.org/10.1145/1167473.1167488

[9] 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. https://doi.org/{10.1007/978-3-642-00768-2_16}

[10] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS '09*. 174–177.

[11] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *SMT '09*. 1–5. https://doi.org/10.1145/1670412.1670413

[12] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE '20*. 1459–1470. https://doi.org/10.1145/3377811.3380398

[13] W.H. Burkhardt. 1967. Generating test programs from syntax. In *Computing*. 53–73.

[14] 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.

[15] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00*. 268–279. https://doi.org/10.1145/351240.351266

[16] SAT Competition. 2023. The International SAT Competition Web Page. Retrieved 2023-10-23 from https://github.com/AdaCore/spark2014

[17] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[18] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *JACM* (2005), 365–473. https://doi.org/10.1145/1066100.1066102

[19] Jonas Duregard, Patrick Jansson, and Meng Wang. 2012. FEAT: Functional Enumeration of Algebraic Types. In *Haskell '12*. 61–72. https://doi.org/10.1145/2430532.2364515

[20] 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. 2021. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation* (2021). https://doi.org/10.1007/s12532-020-00194-3

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI '05*. 213–223. https://doi.org/{10.1145/1064978.1065036}

[22] 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.

[23] K. V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9, 4 (1970), 242–257.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. https://doi.org/10.5555/2362793.2362831

[25] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (apr 2002), 256–290. https://doi.org/10.1145/505145.505149

[26] D. Jackson and C.A. Damon. 1996. Elements of style: analyzing a software design feature with a counterexample detector. *TSE '96* (1996), 484–495. https://doi.org/10.1109/32.538605

[27] 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. https://doi.org/10.1145/3368089.3409763

[28] Rudy Matela Braquehais. 2017. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing*. Ph. D. Dissertation. University of York.

[29] Aina Niemetz, Mathias Preiner, and Clark Barrett. 2022. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *CAV '22*. 92–106. https://doi.org/10.1007/978-3-031-13188-2_5

[30] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-based API testing for SMT solvers. In *SMT '17*. 10.

[31] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. In *OOPSLA '21*. 1–19. https://doi.org/10.1145/3485529

[32] David J. Pearce. 2019. On declarative rewriting for sound and complete union, intersection and negation types. *Journal of Computer Languages* 50 (2019), 84–101. https://doi.org/10.1016/j.jvlc.2018.10.004

[33] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021). https://doi.org/10.1145/3443420

[34] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Haskell '08*. 37–48. https://doi.org/10.1145/1411286.1411292

[35] Neha Rungta. 2022. A Billion SMT Queries a Day (Invited Paper). In *CAV '22*. 3–18. https://doi.org/10.1007/978-3-031-13185-1_1

[36] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning. In *CAV '20*. 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

[37] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California at Berkeley. https://doi.org/10.1007/978-3-642-10672-9_3
[38] SPEC. 2023. SPEC's Benchmarks and Tools. Retrieved 2023-10-23 from https://www.spec.org/benchmarks.html
[39] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. 2004. Software Assurance by Bounded Exhaustive Testing. In *ISSTA '04*. 133–142. https://doi.org/10.1145/1013886.1007531
[40] Geoff Sutcliffe. 2016. The CADE ATP System Competition — CASC. *AI Magazine* (Jul. 2016), 99–101. https://ojs.aaai.org/index.php/aimagazine/article/view/2620
[41] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI '14*. 530–541. https://doi.org/10.1145/2594291.2594340
[42] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusal Effectiveness of Type-Aware Operator Mutation. *OOPSLA '20*, 1–25. https://doi.org/10.1145/3428261
[43] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. *PLDI '20*, 718––730. https://doi.org/10.1145/3385412.3385985
[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI '11*. 283–294. https://doi.org/10.1145/1993316.1993532
[45] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *ISSTA '21*. 322–335. https://doi.org/10.1145/3460319.3464803
[46] 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. https://doi.org/10.1145/3468264.3468540
[47] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI '17*. 347–361. https://doi.org/10.1145/3140587.3062379

Received 2024-04-06; accepted 2024-08-18