Validating SMT Solvers via Semantic Fusion

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

Chengyu Zhang*
Software Engineering Institute
East China Normal University, China
dale.chengyu.zhang@gmail.com

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

Abstract

We introduce *Semantic Fusion*, a general, effective methodology for validating Satisfiability Modulo Theory (SMT) solvers. Our key idea is to *fuse* two existing equisatisfiable (*i.e.*, both satisfiable or unsatisfiable) formulas into a new formula that combines the structures of its ancestors in a novel manner and preserves the satisfiability by construction. This fused formula is then used for validating SMT solvers.

We realized *Semantic Fusion* as YinYang, a practical SMT solver testing tool. During four months of extensive testing, YinYang has found 45 *confirmed, unique bugs* in the default arithmetic and string solvers of Z3 and CVC4, the two state-of-the-art SMT solvers. Among these, 41 have already been fixed by the developers. The majority (29/45) of these bugs expose critical soundness issues. Our bug reports and testing effort have been well-appreciated by SMT solver developers.

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

Keywords: Semantic fusion, SMT solvers, Fuzz testing

ACM Reference Format:
Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3385412.3385985

1 Introduction

Satisfiability Modulo Theory (SMT) solvers check the satisfiability of first-order logic formulas with functions from different theories, such as the booleans, linear and nonlinear arithmetic, unicode strings, *etc.* They are important tools for many programming languages advances and applications, *e.g.*, symbolic execution [12, 20], program synthesis [31], solver-aided programming [33], and program verification [16, 17]. SMT solvers' satisfiability decisions are critical, and incorrect decisions (*i.e.*, soundness bugs) can invalidate the results of their client applications.

Z3 [15] and CVC4 [2] are two state-of-the-art SMT solvers that have been consistently developed for more than ten years. Researchers and practitioners value their extensive theory support and trust the SMT solvers' results. This is justified since soundness bugs in Z3 and CVC4 are rare, and both solvers have extensive regression test suites. However, like any complex software systems, SMT solvers can still have bugs. In fact, almost all critical SMT solver bugs in Z3 and CVC4 have been uncovered directly by their client applications. Such bugs frustrate application developers, and can be catastrophic in safety-critical domains. Besides regression testing, fuzzing has been used to validate SMT solvers.

In 2009, Brummayer and Biere [7] proposed a grammar-based fuzzer called FuzzSMT, which found several bugs in CVC3 (CVC4's predecessor) and early versions of Z3. Within the last ten years, SMT solvers have greatly matured, and finding bugs in them has become more difficult. More recent efforts on testing SMT solvers by fuzzing [5, 9] have targeted the unicode string theory and found a few bugs in Z3's string solvers. Yet none has targeted other SMT theories nor found bugs in recent versions of CVC4.

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

1. *Formula Concatenation*: Concatenate φ_1 and φ_2 by formula conjunction or disjunction;
2. *Variable Fusion*: Create fresh variables to connect the variable sets of φ_1 and φ_2 using *fusion functions*; and
3. *Variable Inversion*: Substitute some occurrences of the chosen variables in φ_1 and φ_2 by *inversion functions*.

Figure 1 illustrates *Semantic Fusion* on two satisfiable formulas φ_1 and φ_2. We first concatenate φ_1 and φ_2, and obtain φ_{concat} as a result. Then, we introduce a fresh variable z and a *fusion function* $f(x, y) = x + y$, and construct a relation $z = f(x, y)$, which induces two equations $x = z - y$ and

*Both authors contributed equally to this work.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385985

$\varphi_1 = x > 0 \wedge \boxed{x} > 1$
$\varphi_2 = \boxed{y} < 0 \wedge y < 1$
$\varphi_{concat} = (x > 0 \wedge \boxed{x} > 1) \wedge (\boxed{y} < 0 \wedge y < 1)$
$\varphi_{fused} = (x > 0 \wedge \boxed{z - y} > 1) \wedge (\boxed{z - x} < 0 \wedge y < 1)$

Figure 1. *Semantic Fusion* on two satisfiable formulas φ_1 and φ_2. Variable z realizes the fusion function $z = x + y$. Shaded: randomly chosen occurrences of x and y to be replaced by variable inversion terms: $z - y$ for x and $z - x$ for y.

$y = z - x$. From these two equations, we obtain two *inversion functions* $r_x(y, z) = z - y$ and $r_y(x, z) = z - x$. Next, we replace the highlighted occurrences of x and y by the corresponding inversion functions $r_x(y, z)$ and $r_y(x, z)$, which results in formula φ_{fused}. By construction, the formula φ_{fused} is also satisfiable. We feed φ_{fused} to the SMT solver under test and observe the result. If the result is `unsat`, we have detected a (soundness) bug in the SMT solver under test.

Bug Hunting with YinYang. We have engineered a practical realization of *Semantic Fusion*, which we call YinYang. During only four months of testing, we have used YinYang to find and report 57 bugs in the default arithmetic and string solvers of Z3 and CVC4. Out of these, 45 were confirmed and 41 were fixed by the developers. We have found 29 bugs that expose soundness issues, the most critical type of bugs in SMT solvers. YinYang found 24 such soundness bugs in Z3 in multiple logics. It found more than 2/3 of all soundness bugs in the nonlinear logic in Z3 since 2015, and around 1/3 of all soundness bugs in its string logics. In CVC4, YinYang detected 5 soundness bugs; 4 were labeled “major” by the CVC4 developers — there were only 9 other such bugs in CVC4’s bug tracker. To the best of our knowledge, none of the previous approaches has ever found a soundness bug in CVC4. The SMT solver developers appreciated our testing effort and bug reports, and specifically commented "great find!", "excellent find!", "nice catch!", *etc.*

Main Contributions.

- We introduce *Semantic Fusion*, a novel, general, principled methodology for stress-testing SMT solvers;
- Based on the *Semantic Fusion* methodology, we design and develop the first highly effective tool, YinYang, for SMT solver validation — the tool is customizable and conveniently supports various SMT theories;
- We conduct a four-month extensive testing of Z3 and CVC4 using YinYang to demonstrate its effectiveness — we have found and reported a total of 57 bugs with 45 confirmed and 41 fixed in their default arithmetic and string solvers, the largest and most successful testing campaign against modern SMT solvers; and
- We present several in-depth evaluations to understand YinYang’s effectiveness in terms of improved code coverage and with respect to a survey of the historic bugs in the SMT solvers Z3 and CVC4.

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

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

Figure 2. Formulas φ_1 and φ_2 in the SMT-LIB format. Shaded: variables to be replaced by inversion function terms.

Paper Organization. The rest of the paper is structured as follows. Section 2 illustrates the high-level idea behind *Semantic Fusion* via two examples. Section 3 formalizes our *Semantic Fusion* approach and describes the implementation of YinYang. Next, we give details on our extensive evaluation (Section 4) and show sampled bugs to highlight the diverse types of bugs that YinYang can find (Section 4.3). Finally, we survey related work (Section 5) and conclude (Section 6).

2 Illustrative Examples

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

SMT-LIB Language. The SMT-LIB language is the current standard input language for SMT solvers [4]. We focus on the following statements of the SMT-LIB language: `declare-fun`, `declare-const`, `define-fun`, `assert`, `check-sat`. Variables are declared as zero-valued functions. For example, the statement "`(declare-fun a () Real)`" declares a variable of type real. An `assert` statement specifies constraints. The predicates within the asserts can be of mixed types, *e.g.*, the assert "`(assert (<= (/ x 4) (* 5 x)))`" includes predicates of real and boolean types. Operations are specified in the prefix notation. Multiple asserts can be viewed as the conjunction of the constraints in each individual assert statement. The `check-sat` statement queries the solver to decide on the satisfiability of a formula. If all constraints are satisfied, the formula is satisfiable; otherwise, the formula is unsatisfiable.

2.1 SAT Fusion

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

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

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

Figure 3. Fused formula φ_{sat} in the SMT-LIB format. It triggered a soundness bug in CVC4.
https://github.com/CVC4/CVC4/issues/3413

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

Step 1: Formula Conjunction: We conjoin formula φ_1 with formula φ_2 and obtain $\varphi_1 \wedge \varphi_2$ as a result. In the SMT-LIB format, this conjunction can be carried out by simply merging the variable declaration and assert blocks.

Step 2: Variable Fusion: We introduce a fresh variable z to fuse the integer variable pairs x in φ_1 and y in φ_2. We define a fusion function: $f(x, y) = x \cdot y$ and construct an equation $z = f(x, y)$. The choice of the fusion function f is determined by the type of the fused variables (*cf.* Section 3). We fuse the occurrences of variables x and y.

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

Why is φ_{sat} satisfiable? Intuitively, because we can construct a model for φ_{sat} from models for φ_1 and φ_2. Let M_1 be a model for φ_1 and M_2 be a model for φ_2. We construct M for φ_{sat} with $M = M_1 \cup M_2 \cup \{z \mapsto M_1(x) \cdot M_2(y)\}$ (see Section 3 for details). Formula φ_{sat} in Figure 3 is a real case. It triggered a soundness bug in CVC4, which made CVC4 incorrectly report `unsat` on φ_{sat}. We reported this issue to the GitHub CVC4's issue tracker. As per the developers, this was a regression introduced by recent code changes, and they promptly fixed the bug.

2.2 UNSAT Fusion

UNSAT fusion combines two unsatisfiable formulas into an unsatisfiable formula. We describe the idea behind UNSAT

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

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

Figure 4. Formulas φ_3 and φ_4 in the SMT-LIB format. Shaded: variables to be replaced by inversion function terms.

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

Figure 5. Fused formula φ_{unsat} of φ_3 and φ_4 that triggered a soundness bug in Z3.
https://github.com/Z3Prover/z3/issues/2391

fusion in four steps: (1) Formula Disjunction, (2) Variable Fusion, (3) Variable Inversion, and (4) Adding Fusion Constraints. While steps (1), (2) and (3) are similar as those in SAT fusion, UNSAT fusion needs an additional fourth step to ensure the unsatisfiability of the fused formula.

Consider the formulas φ_3 and φ_4 in Figure 4:

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

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

Step 4: Add Fusion Constraints: We add $z = f(x, y)$, $x = r_x(y, z)$ and $y = r_y(x, z)$ to the fused formula. We call them *fusion constraints.* Since random substitutions may render the fused formula satisfiable, we need the fusion constraints to ensure that r_x and r_y recover x and y (see Section 3 details). The SMT-LIB code of the resulting fused formula is shown in Figure 5.

The fused formula φ_{unsat} in Figure 5 has triggered a soundness bug in Z3, *i.e.*, Z3 reports `sat` on φ_{unsat}, which is incorrect since the formula is unsatisfiable by construction. This bug is only triggered by the fused formula; it cannot be triggered by either of the seed formulas nor by the disjunction/conjunction of the two seed formulas φ_3 and φ_4.

3 Approach

This section presents *Semantic Fusion* and how we apply it to stress-testing SMT solvers. We propose two approaches, *SAT Fusion* and *UNSAT Fusion*, and prove their correctness. We describe our tool YinYang, a practical implementation of SAT Fusion and UNSAT Fusion.

3.1 Definitions

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

3.2 Semantic Fusion

The insight of Semantic Fusion is to combine two seed tests into a new test that fuses the structures of its ancestors. Applying this to two formulas φ_1 and φ_2 of same satisfiability, we fabricate a fused formula φ_{fused} that is equisatisfiable.

Definition 1 (Fusion function)**.** *Let φ_1, φ_2 be formulas, $x \in vars(\varphi_1)$, and $y \in vars(\varphi_2)$. Let z be a fresh variable $z \notin vars(\varphi_1) \cup vars(\varphi_2)$. We define*

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

Function f is called a ***Fusion function****.*

Having defined fusion function, we next invent inversion functions to recover the original values for x and y.

Definition 2 (Inversion function)**.** *Let φ_1 and φ_2 be formulas and f be a fusion function. For $x \in vars(\varphi_1)$, $y \in vars(\varphi_2)$ and $z = f(x, y)$, we define*

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

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

As an example, consider the fusion function $f(x, y) = x + y$. The corresponding inversion functions for x and y are: $r_x(y, z) = z - y$, and $r_y(x, z) = z - x$. We next present a proposition that shows how we fuse two satisfiable formulas into an equisatisfiable formula.

Proposition 1 (SAT fusion)**.** *Let φ_1, φ_2 be satisfiable formulas with $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. Let further $x \in vars(\varphi_1)$ $y \in vars(\varphi_2)$ be variables. Then, the formula*

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

is satisfiable.

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

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

Since $x = r_x(y, z)$ by Definition 2, $M(x) = M(r_x(y, z))$. Thus, $M(\varphi_1[r_x(y, z)/x]_R) = M(\varphi_1)$ via structural induction. By M's construction, $M(\varphi_1) = M_1(\varphi_1)$, thus $M \models \varphi_1[r_x(y, z)/x]_R$. Similarly, $M \models \varphi_2[r_y(x, z)/y]_R)$, and hence $M \models \varphi_{sat}$. □

Proposition 1 enables us to fuse two satisfiable formulas and obtain a satisfiable formula as a result. We would also like to fuse unsatisfiable formulas into an unsatisfiable formula. However, we cannot simply fuse two unsatisfiable formulas using Proposition 1 as the following counterexample shows. Consider the unsatisfiable formulas $\varphi_1 = x > 0 \wedge x < 0$, $\varphi_2 = y \neq y$, and the fusion function $z = x + y$. If we replace the shaded occurrence of x by $y - z$ and y by $x - z$, we get the following formula: $(x > 0) \wedge (z - y < 0) \wedge (z - x \neq y)$. This is a satisfiable formula, *e.g.*, any assignments for x, y and z that satisfy $x > 0$ and $y > z$ realize a model. The problem here is that we can freely choose z that does not necessarily preserve $z = f(x, y)$. To prevent this, we add the constraint $z = f(x, y)$ to the formula. For fusing unsatisfiable formulas, we disjoin the formulas, since this is likely to increase the effort of SMT solvers to prove the formula unsatisfiable.

Proposition 2 (UNSAT fusion)**.** *Let φ_1, φ_2 be unsatisfiable formulas with $vars(\varphi_1) \cap vars(\varphi_2) = \emptyset$. Let further $x \in vars(\varphi_1)$, $y \in vars(\varphi_2)$ be variables. Then, the formula*

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

is unsatisfiable.

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

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

Say $\varphi_1[r_x(y, z)/x]_R \wedge z = f(x, y)$ were satisfiable. The formula $\varphi_1[r_x(y, z)/x]_R \wedge z = f(x, y)$ is equivalent to the formula $\varphi_1[r_x(y, z)/x]_R[f(x, y)/z] \wedge z = f(x, y)$, which, by Definition 2, is equivalent to $\varphi_1 \wedge z = f(x, y)$. This contradicts

Type	**Fusion Function**	**Variable Inversion Functions**	
		r_x	r_y
Int	$x + y$	$z - y$	$z - x$
	$x + c + y$	$z - c - y$	$z - c - x$
	$x * y$	$z \; div \; y$	$z \; div \; x$
	$c_1 * x + c_2 * y + c_3$	$(z - c_2 * y - c_3) \; div \; c_1$	$(z - c_1 * x - c_3) \; div \; c_2$
Real	$x + y$	$z - y$	$z - x$
	$x + c + y$	$z - c - y$	$z - c - x$
	$x * y$	z/y	z/x
	$c_1 * x + c_2 * y + c_3$	$(z - c_2 * y - c_3)/c_1$	$(z - c_1 * x - c_3)/c_2$
String	x *str++* y	*str.substr* z 0 (*str.len* x)	*str.substr* z (*str.len* x) (*str.len* y)
	x *str++* y	*str.substr* z 0 (*str.len* x)	*str.replace* z x ""
	x *str++* c *str++* y	*str.substr* z 0 (*str.len* x)	*str.replace* (*str.replace* z x "") c ""

Figure 6. Variable fusion functions with their corresponding variable inversion functions categorized by types Int, Real and String. The coefficients $c_1, \cdots, c_3$ are randomly chosen, and div denotes integer division.

the assumption, *i.e.*, the unsatisfiability of φ_1. The case for $\varphi_2[r_y(x,z)/y]_R \wedge z = f(x,y)$ is symmetric. □

Proposition 2 enables us to fuse two unsatisfiable formulas into an unsatisfiable formula and complements Proposition 1. We could also apply fusion to mixed formula pairs, *i.e.*, when φ_1 is satisfiable and φ_2 is unsatisfiable. We can use $\varphi_1[r_x(y,z)/x] \vee \varphi_2[r_y(x,z)/y]$ for a satisfiable fused formula and $\varphi_1[r_x(x,z)/x] \wedge \varphi_2[r_y(x,z)/y] \wedge z = g(x,y)$ for an unsatisfiable fused formula.

3.3 Fusion and Inversion Functions

We now give exemplary fusion and inversion functions (see Figure 6) and explain the intuitions behind them. Let us consider the Int and Real categories. The first two fusion and inversion functions in these categories are based on addition/subtraction and multiplication/division. When division and multiplication of variables are used as function and inversion functions, a formula in linear logic might become non-linear. This is because we replace free variables occurrences by variable inversion functions that include the division operator. Another inversion function for real and integer arithmetic is $c_1 * x + c_2 * y + c_3$. The intuition behind $c_1 * x + c_2 * y + c_3$ is to synthesize arbitrary polynomial combinations of the variables x and y since $c_1, \cdots, c_3$ are random coefficients. Let us consider Strings next. In the first row of the String category, we define z as the concatenation of the two strings x and y. Say x = "foo" and y = "bar", then $z = x$ *str++* y = "foobar". We retrieve x by the substring of z from 0 to $|x|$, for y the substring from $|y|$ to the end of z. Another way to retrieve y is to use the replace function instead of substring. The expression *str.replace* z x "" denotes the replacement of the first occurence of x in z by the empty string "", which results in "bar".

In addition, we can insert a random string c into x *str++* y by x *str++* c *str++* y to make the fusion function more complex, and then retrieve y by replacing x and c with "" sequentially. We emphasize that *Semantic Fusion* is not restricted to these fusion and inversion functions of Figure 6. A richer set of fusion and inversion functions can be designed based on the generic Definitions 1 and 2.

3.4 YinYang

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

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

Algorithm 2 presents the implementation of the `fuse` function. It takes two seed formulas φ_1 and φ_2 as input and retrieves the sets of their free variables $vars(\varphi_1)$ and $vars(\varphi_2)$, respectively. Then, we create random triplets T, for $(z, x, y) \in T$ where $x \in vars(\varphi_1)$ and $y \in vars(\varphi_2)$, and z

Algorithm 1: YinYang's main process

Procedure YinYang (o, S, Φ_o):
  $incorrects \leftarrow \emptyset,\ crashes \leftarrow \emptyset$
  while *no termination criterion met* **do**
    $\varphi_1 \leftarrow$ `random.choice(`Φ_o`)`
    $\varphi_2 \leftarrow$ `random.choice(`Φ_o`)`
    $\varphi_{fused} \leftarrow$ `fuse(`o, φ_1, φ_2`)`
    if $S(\varphi_{fused}) = crash$ **then**
      $crashes \leftarrow crashes \cup \{\varphi_{fused}\}$
    else if $S(\varphi_{fused}) \neq o$ **then**
      $incorrects \leftarrow incorrects \cup \{\varphi_{fused}\}$

Algorithm 2: *Semantic Fusion* on two SMT formulas

1 **Function** `fuse(`o, φ_1, φ_2`)`:
2   $vars(\varphi_1) \leftarrow$ `free_variables(`φ_1`)`
3   $vars(\varphi_2) \leftarrow$ `free_variables(`φ_2`)`
4   $T \leftarrow$ `random_map` $(vars(\varphi_1), vars(\varphi_2))$
5   $\varphi_1', \varphi_2' \leftarrow$ `variable_fusion` $(T, \varphi_1, \varphi_2)$
6   **if** $o =$ `sat` **then**
7   **return** $\varphi_1' \wedge \varphi_2'$
8   **else**
9     $\varphi' \leftarrow \varphi_1' \vee \varphi_2'$
10   **foreach** $(z, x, y) \in T$ **do**
11       $\varphi' \leftarrow \varphi' \wedge z = f(x, y)$
12   **return** φ'
13 **Function** `variable_fusion(`T, φ_1, φ_2`)`:
14   $\varphi_1' \leftarrow \varphi_1,\ \varphi_2' \leftarrow \varphi_2$
15   **foreach** $(z, x, y) \in T$ **do**
16     $\varphi_1' \leftarrow \varphi_1'[r_x(y, z)/x]_R$
17     $\varphi_2' \leftarrow \varphi_2'[r_y(x, z)/y]_R$
18   **return** φ_1', φ_2'

is the fresh variable. In the `variable_fusion` function, we substitute randomly chosen occurrences of x in φ_1 and y in φ_2 by the inversion function terms $r_x(y, z)$ and $r_y(x, z)$ from Table 6. If oracle o is *sat*, we perform SAT fusion (Proposition 1) and return the conjunction of φ_1' and φ_2' directly (Line 7). If oracle o is *unsat*, we perform UNSAT fusion, *i.e.*, we disjoin φ_1' and φ_2', and add a fusion constraint for each triplet $(x, y, z) \in T$ (Lines 9-11) and return the result.

In principle, YinYang guarantees the absence of false positives, given that the seed formulas Φ_o are correctly labeled. In practice, the solvers may report *unknown*, which could be either seen as a crash or ignored.

Implementation of YinYang. We implemented YinYang in a total of 1,032 lines of Python 3.7 code. YinYang is able to run in multiple-threaded mode, which significantly increases its throughput. Users can customize the command-line interface of YinYang for customized SMT solvers and/or features. YinYang accepts SMT solver binaries as test targets and obtains the solving results from the stdout stream, which makes YinYang compatible with most SMT solvers. A lightweight SMT-LIB v2 parser is implemented for getting free variables and assertions. The formula concatenation and variable substitution are implemented by string operations, which makes YinYang compatible to most of the formulas without additional implementation. When there are multiple fusion/inversion functions choices for f, r_x and r_y, YinYang makes a random choice.

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

Figure 7. The formula counts of the respective benchmarks.

4 Empirical Evaluation

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

Result Summary and Highlights.

- *Many confirmed bugs:* In four months, YinYang found 45 unique bugs in Z3 and CVC4. Out of these, 41 were already fixed by the developers.
- *Many soundness bugs:* YinYang found 24 soundness bugs in Z3 and 5 in CVC4. These represent 16% of the reported Z3 soundness bugs of the last five years and 11% of the reported CVC4 soundness bugs of the last nine years. Some of the bugs affect multiple historical release versions.
- *Bugs in various logics:* YinYang found bugs in various logics, *e.g.*, QF_NRA, QF_NIA, NRA, NIA, QF_S, and QF_SLIA. Most of the bugs in Z3 were found in NRA (15) and QF_S (15), while most of the bugs in CVC4 were found in QF_S (4).

Status	Z3	CVC4	Total
Reported	44	13	57
Confirmed	37	8	45
Fixed	35	6	41
Duplicate	4	1	5
Won't fix	2	0	2

(a)

Type	Z3	CVC4	Total
Soundness	24	5	29
Crash	11	1	12
Performance	1	2	3
Unknown	1	0	1

(b)

Logic	Z3	CVC4	Total
NIA	2	1	3
NRA	15	1	16
QF_NIA	0	1	1
QF_NRA	2	0	2
QF_S	15	4	19
QF_SLIA	3	1	4

(c)

Figure 8. (a) Status of the reported bugs in Z3 and CVC4, (b) Types of the confirmed bugs in Z3 and CVC4, and (c) Affected SMT logics of the confirmed bugs in Z3 and CVC4.

4.1 Evaluation Setup

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

Test Seed Formulas. Figure 7 shows the formula counts of the respective benchmarks that we used. The majority of the seed formulas come from the SMT-LIB benchmark suite maintained by the SMT-LIB Initiative [3]. We chose the SMT-LIB benchmarks as our test seeds since they make the largest collection of SMT formulas in the SMT-LIB 2.6 language [4]. The SMT-LIB benchmarks are also used in the SMT Competition [14]. Therefore, these formulas are unlikely to trigger bugs in Z3 and CVC4 since they have already been run on them. This helps us isolate the effects of *Semantic Fusion*. We choose the following logics: LIA, LRA, NRA, QF_LIA, QF_LRA, QF_NRA, QF_SLIA, and QF_S. L represents linear, N represents non-linear, IA represents integer arithmetic, RA represents real arithmetic, QF represents quantifier-free, and S represents string logic.

Besides the SMT-LIB benchmarks, we also used the benchmarks from StringFuzz [19]. The formulas from the StringFuzz benchmarks are the in QF_S logic and in the SMT-LIB 2.6 language. They do not trigger any bugs in the latest versions of Z3 and CVC4. We preprocessed all formulas (from the SMT-LIB benchmarks and StringFuzz) with Z3 to subdivide them into a satisfiable and an unsatisfiable set. We cross-checked with CVC4 to ensure the correctness of these ground truths. In total, we obtained $75,097$ seed formulas, $46,760$ of which are satisfiable and $28,337$ are unsatisfiable.

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

- Z3 ($5,196$ stars on GitHub) and CVC4 (361 stars on GitHub) are the two most popular SMT solvers. They are mature and widely-used in academia and industry.
- Z3 and CVC4 have state-of-the-art performance. Both regularly rank high in the annual SMT competitions [14].
- Z3 and CVC4 support most of the features and logics in the SMT-LIB standard, while the other SMT solvers only partially support the SMT-LIB standard.
- Z3 and CVC4 have open source issue trackers on GitHub, and their developers are active and responsive. This helps our testing effort as we can quickly get feedback on our bug reports, and filed bugs are fixed promptly.

For CVC4, we use its `-strings-exp` option to enable support for the string logic and default configuration for the other logics. For Z3, we use `smt.string_solver=z3str3`, its default configuration for string logic, and default configuration for the other logics. We compiled both solvers with assertions enabled.

Bug Reduction. When a bug is found, we need to reduce the fused formula to a small enough size for reporting. We use C-Reduce [29], a C code reduction tool, which also works for the SMT-LIB language. We implemented a pretty printer to help with the bug reduction process, *i.e.*, when C-Reduce has converged to a still very large formula or hangs. The pretty-printer makes simple modifications to the AST of a formula, *i.e.*, flattens nestings of the same operator, removes additions and multiplications with neutral elements and returns the modified formula in a human-readable format.

4.2 Quantitative Evaluation

We guide our quantitative evaluation by four consecutive research questions.

RQ1: How many bugs can YinYang find?

From July 2019 to October 2019, we extensively tested Z3 and CVC4 with YinYang. YinYang usually reports many bug-triggering test cases in one testing round. To avoid duplicate bug reports, we always use the trunk versions of the solvers for testing. Once the developers have fixed a bug, we validate the fixed version on the rest of the formulas which triggered bugs in the previous testing round. If the solvers passed all formulas and no bug was triggered, we started a new testing round. During our four months of testing, YinYang generated around 800 million test formulas. On average, YinYang generates 41.5 test formulas per second when run in the single-threaded mode. Figure 8a shows the bug counts categorized by reported, confirmed, fixed, duplicate and won't fix. From the 57 reported bugs, 45 bugs were confirmed by the developers as real bugs and 41 bugs were fixed. Although we devoted equal testing effort to both solvers, YinYang found more bugs in Z3 (37 confirmed bugs) and clearly fewer bugs in CVC4 (8 confirmed bugs). Having observed that YinYang can find a significant number of bugs in Z3 and CVC4, Figure 8b shows the bug type overview of YinYang's findings. For bug reporting, we distinguish the following three types of bugs:

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

Overall, the most common bug category is for soundness bugs (29 out of the 57 reported bugs) followed by crash bugs (12 out of the 57 reported bugs). This is consistent for both solvers, which shows the strength of YinYang in finding soundness bugs. Although we designed YinYang to target soundness and crash bugs, we also considered performance bugs. We have found these bugs during the reduction process of C-Reduce. As performance bugs are less interesting than soundness and crash bugs, we stopped reporting performance bugs after several bug reports and solely focused on soundness and crash bugs subsequently. Figure 8c shows the logic distribution among the confirmed bugs. In Z3, we found most of the bugs in NRA (15) followed by QF_S (15), QF_SLIA (3), NIA (2) and QF_NRA (2). In CVC4, we found most of the bugs in QF_S (4).

RQ2: How significant are the bug-finding results?

To approach this question, we consider the most critical bugs in SMT solvers, *i.e.*, the soundness bugs. Soundness bugs in SMT solvers are rare and heavily penalized when detected in the SMT competitions [14]. We have conducted a study on soundness bugs based on the GitHub issue trackers of Z3 and CVC4. The results are shown in Figure 9. For Z3, we considered April 2015 as the start date, right after Z3 was released on GitHub. For CVC4, we have data since July 2010 as CVC4's previous Bugzilla issue tracker was migrated to GitHub. Z3 supports a myriad of logics and has become very popular on GitHub (5,196 stars). However, there were only 146 soundness bugs reported on the Z3 issue tracker from April 2015 to October 2019. For CVC4 this number is even lower. Since July 2010, there were only 42 soundness bugs. Of all the soundness bugs in Z3, we found 24 out of 146

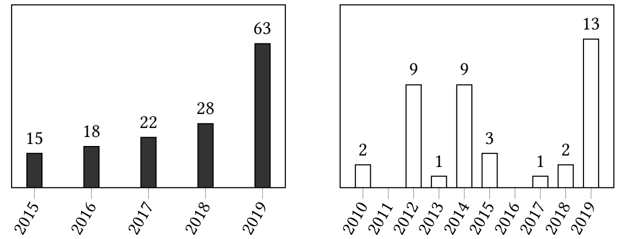

Figure 9. Number of soundness bugs in Z3 (left) and CVC4 (right) per year.

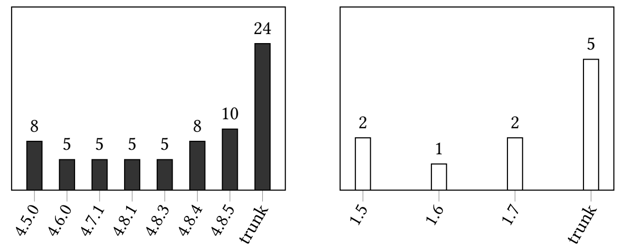

Figure 10. Number of found soundness bugs that affect corresponding release versions of Z3 (left) and CVC4 (right).

(16%). For CVC4, we found 5 soundness bugs out of 43 (11%) in only four months. We found 18 out of the 25 soundness bugs in non-linear logics in Z3 since 2015 and 15 out of the 53 soundness bugs in its string logic. As an intermediate conclusion to RQ2, we state that YinYang has found a significant number of the soundness bugs in both Z3 and CVC4. To more deeply understand the significance of our soundness bug findings, we studied the influence of soundness bugs in different releases of Z3 and CVC4. Figure 10 shows the results. We selected all released versions of Z3 and CVC4 that support the formulas triggering soundness bugs. Z3 4.5.0 was released on November 8, 2016, and CVC4 1.5 was released on July 10, 2017, which means that YinYang found 8 soundness bugs in Z3 that were latent for 3 years, and 2 soundness bugs in CVC4 that were latent for 2 years. YinYang has found long-latent bugs missed by solver developers, users, regression testing, and prior automated testing. This confirms the significance of our bug findings.

RQ3: Can YinYang improve code coverage?

In this research question, we use code coverage, a standard evaluation metric for software testing, to understand whether YinYang can cover additional code inside the SMT solvers. To investigate the coverage improvement of YinYang, we consider the following steps:

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

		LIA						LRA						NRA					
		SAT			UNSAT			SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	10.6	14.6	3.4	9.4	13.2	2.8	10.4	14.5	3.3	9.8	13.8	3.1	-	-	-	10.6	13.1	3.5
	YinYang	12.0	16.7	3.8	14.5	18.6	4.9	13.4	17.3	4.3	13.3	16.8	4.3	-	-	-	11.9	14.8	3.9
CVC4	*Benchmark*	14.7	29.1	5.4	15.3	28.9	6.0	14.2	27.3	5.3	12.8	24.9	4.6	-	-	-	16.0	30.9	6.2
	YinYang	16.4	31.4	6.2	17.5	31.9	7.2	16.1	30.6	6.4	15.7	29.4	6.2	-	-	-	17.9	33.9	7.2

		QF_LIA						QF_LRA						QF_NRA					
		SAT			UNSAT			SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	11.5	16.3	3.7	13.3	17.6	4.3	7.8	12.6	2.6	7.3	11.1	2.4	12.0	13.5	4.6	11.3	13.0	4.1
	YinYang	14.8	19.8	5.1	16.1	20.6	5.5	14.7	18.7	5.2	14.3	17.8	5.2	13.4	15.3	5.1	12.5	14.5	4.6
CVC4	*Benchmark*	11.0	20.2	3.5	11.6	20.2	3.8	10.7	19.9	3.3	10.6	19.6	3.4	12.8	22.6	4.7	13.4	22.6	5.0
	YinYang	12.7	23.8	4.4	12.3	20.9	4.1	13.7	27.2	5.1	12.9	25.6	4.6	15.2	28.7	5.8	15.7	27.8	6.2

		QF_SLIA						QF_S						StringFuzz					
		SAT			UNSAT			SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
Z3	*Benchmark*	10.2	15.2	3.4	11.5	15.4	4.0	12.5	17.7	4.3	11.8	16.2	4.0	13.4	18.3	4.6	12.7	17.5	4.2
	YinYang	10.4	15.5	3.5	13.1	16.9	4.7	12.6	17.8	4.3	12.8	17.0	4.4	13.7	18.4	4.8	13.6	18.3	4.7
CVC4	*Benchmark*	15.2	25.3	6.8	18.4	32.6	7.4	16.2	28.6	6.5	14.5	26.2	5.6	19.6	36.3	8.2	20.3	35.9	9.1
	YinYang	16.2	26.5	7.3	19.6	34.0	8.0	16.6	29.9	6.7	14.8	26.3	5.9	20.0	36.6	8.5	20.8	36.2	9.5

Figure 11. Code coverage evaluations. The numbers represent the percentage (%) coverage for the corresponding coverage metric. Column *l*, *f*, *b* represent line coverage, function coverage, and branch coverage respectively. Higher coverage between *Benchmark* and *YinYang* is shaded.

The timeout for the solvers is set to two seconds. For measuring coverage, Z3 and CVC4 are compiled in debug mode and without optimizations. The coverage measurement tool is Gcov [1]. Table 11 shows the results. The numbers represent the percentages (%) of lines (l), functions (f) and branch (b) coverage respectively. The highest coverages are shaded. Since SMT solvers have large code bases (Z3 has over 436K LOC, CVC4 has over 238K LOC), 1% line coverage improvement translates to thousands of additionally covered lines. First, we observe that both Z3 and CVC4 mostly achieve less than 30% line, function and branch coverage. This may be explained by the many mutually exclusive features that CVC4 and Z3 support. The main observation is that *YinYang* can consistently increase the coverage achieved by the *Benchmark*. This indicates that YinYang can enhance benchmark formulas and exercise previously uncovered code. Furthermore, YinYang can achieve this noticeable coverage improvement in only one hour in the single-threaded mode, showing the effectiveness of YinYang.

RQ4: Is Semantic Fusion necessary for finding bugs?

This research question investigates whether we can obtain our bug findings with a simpler approach. As mentioned earlier, *Semantic Fusion* consists of two main steps: (1) formula concatenation and (2) variable fusion and inversion. Step (2) is the core technique of *Semantic Fusion*. Let ConcatFuzz be the simple concatenation tool where we solely perform step (1) and disable step (2), *i.e.*, ConcatFuzz only combines formulas by conjunction (for satisfiable formulas) and disjunction (for unsatisfiable formulas) without variable fusion and inversion. To see whether *Semantic Fusion* is necessary for triggering bugs, we ran ConcatFuzz on the ancestor seeds of 50 reported bugs that YinYang found. In only 5 out of 50 cases, ConcatFuzz was able to retrigger the bug. This indicates that simple formula concatenation is unable to trigger most of the bugs found by YinYang, which shows the necessity of the core technique of *Semantic Fusion*. In addition, we also repeated the code coverage evaluation of RQ3 to understand the code coverage difference between ConcatFuzz and YinYang. Figure 12 shows the code coverage of ConcatFuzz and YinYang averaged over all logics. The results show that both YinYang and ConcatFuzz consistently achieve higher line, function and branch coverage than

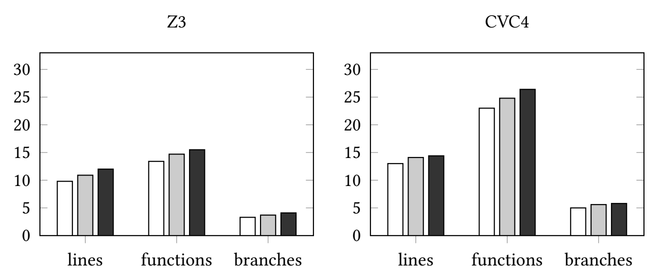

Figure 12. Coverage improvement (%) of ConcatFuzz (in gray) and YinYang (in black) over Benchmark (in white) averaged over all logics.

the Benchmark. The coverage improvement of ConcatFuzz over the benchmark, partially explains why ConcatFuzz can retrigger some of the bugs. However, the results also reflect that the average code coverage of YinYang dominates ConcatFuzz. YinYang achieves on average 1.1% more lines (approx. 2,800 lines) in Z3 and 0.3% (approx. 480 lines) in CVC4, which can partially explain why YinYang can trigger more bugs than ConcatFuzz. In summary, our results show that semantic fusion is necessary for YinYang's effectiveness, and code coverage and bug count correlate.

4.3 Assorted Bug Samples

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

Figures 13a shows an unsatisfiable formula in the string logic QF_S. The formula has two asserts. The second assert demands variable `a` to be the concatenation of `b` and `c`. The first asserts includes the query on whether `c` is matched by regex "aa"*. This is conjoined with a check on whether `c` equals "0" (see lines 8 to 11). The formula is unsatisfiable since the two conditions contradict each other. However, Z3 reports `sat` on Formula 13a, which is incorrect.

Figure 13b also shows a formula in the string logic QF_S. The formula is unsatisfiable but CVC4 incorrectly reported `sat` on it. The formula has been reduced from the same original test case as Formula 13a which also triggered a Z3 soundness bug. CVC4 and Z3 both report `sat` on the original formula. These two bugs show the benefits of our approach over differential testing. In this case, differential testing would not be able to capture these bugs as the results from both solvers are the same, but incorrect. The root cause of the bug is a missed corner case in the *str.to.int* reduction function for an empty string. The bug was labeled as *major* in the CVC4 bug tracker.

Figure 13c shows an unsatisfiable formula in the non-linear real arithmetic logic QF_NRA. Z3 reported *sat* on this formula and gave the following incorrect model:

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

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

Figure 13d shows an unsatisfiable formula in the QF_SLIA logic. CVC4 incorrectly reported `sat` on this formula and gave an incorrect model. The root cause for this bug is an unsound formula simplification of CVC4. The bug is labeled *major* by the CVC4 developers. They rewrote the simplification strategy to fix the bug.

Figure 13e shows an unsatisfiable formula in the string logic QF_S. Z3 incorrectly reported `sat` on this formula and gave an incorrect model. The developers made some major changes to fix this bug — 28 files with 486 additions and 144 deletions were necessary to fix it. The bug is triggered by an incorrect implementations of the `suffixof` and `prefixof` operators.

Figure 13f shows a formula in quantified real arithmetic (NRA). Z3 crashed when solving this formula with the following message:

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

According to the bug fix of the developer, the root cause for this crash was an error in the rewriting strategy for the comparison operators <= and >=.

4.4 Discussion

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

Quality of the bug-finding results. Our 4-month testing effort produced significant, high-quality results. We focused specifically on finding bugs in the default modes of arithmetic and string solvers. As most users invoke SMT solvers in their default modes, such bug reports are thus particularly valuable. YinYang found 29 such critical soundness bugs, which shows the effectiveness of *Semantic Fusion*. In addition, the SMT solver developers greatly appreciated our bug finding effort and reports with comments like "great find!", "excellent find!", "nice catch!", *etc.* All of our CVC4 soundness bug reports were labeled *major*, which are rare on the CVC4 issue tracker — in fact, only 13 soundness bugs in the CVC4 issue tracker are labeled *major*. Our bugs in Z3 have been fixed in the recent release versions, which makes Z3 more reliable and robust.

Limitations and future work. *Semantic Fusion* is shown to be effective for SMT solver testing. It does also come with some limitations. First, *Semantic Fusion* relies on the given

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

(a) *Soundness bug in Z3:* Z3 returns `sat` on this unsatisfiable QF_S formula. The formula and Formula 13b are reduced from the same seed.
https://github.com/Z3Prover/z3/issues/2618

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

(c) *Soundness Bug in Z3:* Z3 reported `sat` on this unsatisfiable QF_NRA formula.
https://github.com/Z3Prover/z3/issues/2391

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

(e) *Soundness bug in Z3:* Z3 reports `sat` on this unsatisfiable QF_S formula.
https://github.com/Z3Prover/z3/issues/2513

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

(b) *Soundness bug in CVC4:* CVC4 reports `sat` on this unsatisfiable QF_S formula. This and the formula in Figure 13a are reduced from the same original formula.
https://github.com/CVC4/CVC4/issues/3357

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

(d) *Soundness bug in CVC4:* CVC4 reports `sat` on this unsatisfiable QF_SLIA formula.
https://github.com/CVC4/CVC4/issues/3203

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

(f) *Crash Bug in Z3:* This NRA formula triggers a segmentation fault in Z3.
https://github.com/Z3Prover/z3/issues/2449

Figure 13. Assorted SMT-LIB formulas that trigger bugs in the SMT solvers Z3 and CVC4.

seed formulas. Second, one needs to manually design the fusion and inversion functions; devising variable fusion and variable inversion functions by hand can be difficult. It would be interesting future work to explore the automatic construction of variable fusion and inversion functions.

5 Related Work

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

SAT/SMT Solver Validation. FuzzSMT [7] is the first effort targeting SMT solver validation. It is based on grammar-based blackbox fuzzing and differential testing. Brummayer and Biere evaluated FuzzSMT on the bit-vector logic and found 16 defects in five solvers, but no soundness bugs in Z3. BtorMBT [26] is a model-based testing tool for Boolector [6], an SMT solver for bit-vectors with arrays. It generates sequences of API calls to exploit the features of the solver. BtorMBT did not find bugs in any mature solvers. The more recent StringFuzz [5] focuses on performance issues in the string logic. It generates test cases by either mutating and transforming the benchmarks, or generating random valid formulas. It found 3 performance and implementation bugs in z3str3 by differential testing. Different from these differential testing based approaches, *Semantic Fusion* tackles the test oracle problem by construction, rather than cross-checking, making it capable of testing solver-specific features.

The recent work of Bugariu and Müller [9] proposed a formula synthesis approach to generating SMT formulas in the string logic with known satisfiability. It generates increasingly-complex formulas via satisfiability-preserving transformations. Several bugs in Z3 were reported, while no reported bugs in CVC4. We instead fuse two formulas and preserve the satisfiability via fusion/inversion functions. A closely-related problem to testing SMT solvers is the testing of SAT solvers. FuzzSAT, CNFuzz and 3SAT [8] randomly generate valid formulas and found 14 bugs in 7 SAT solvers via differential testing.

Our work found 45 confirmed bugs in modern, mature and widely-used SMT solvers, significantly more than any prior work on SMT/SAT solver testing. *Semantic Fusion* is also general and can find bugs in various logics while much prior work focuses on the string logic.

Validation of Program Analyzers. With program analyzers becoming increasingly practical and adopted, it is critical to ensure their reliability [11]. Several recent efforts explored this problem, targeting software model checkers, symbolic execution engines, and various static analyzers. Both Zhang *et al.* [35] and Klinger *et al.* [22] developed approaches to testing software model checkers — the approach by Zhang *et al.* [35] is based on reachability queries, while the approach by Klinger *et al.* [22] is based on differential testing. Kapus *et al.* [21] used random program generation and differential testing to find bugs in symbolic execution engines. Wu *et al.* [34] found bugs in alias analyses via cross-checking with dynamic aliasing information. Bugariu *et al.* [10] proposed an approach for finding soundness and precision bugs in numerical abstract domains. Qiu *et al.* [28] and Pauck *et al.* [27] reported experiences in testing and finding defects in analyzers for Android apps.

Many of these program analyzers, such as software model checkers, symbolic execution engines, and program verifiers, critically rely on SMT solvers. Thus, although our work targets SMT solvers, it also helps improve the reliability of program analyzers.

Metamorphic Testing. The test oracle problem is a long-standing challenge in software testing. Metamorphic testing is a general approach to this problem [13, 30]. Its key idea is to leverage existing tests to construct additional ones with expected results via certain metamorphic relations. For example, the technique of equivalence modulo inputs (EMI) [23] is a notable instance of metamorphic testing for compilers. It constructs equivalent test programs for a seed program with respect to a given input by strategically mutating the seed program. To date, the general EMI-based approach and its variants [24, 32, 36] have found more than 1,600 bugs in GCC and Clang/LLVM. EMI and metamorphic testing in general were also adapted to test shader compilers [18, 25].

The *Semantic Fusion* methodology introduced in this paper is also an instance of metamorphic testing — it generates new test formulas by fusing two existing test cases preserving their oracle. *Semantic Fusion* is a new and highly generic metamorphic testing approach that we successfully applied to SMT solver testing.

6 Conclusion

We have introduced *Semantic Fusion*, a novel, principled testing methodology for validating SMT solvers. The key idea behind *Semantic Fusion* is to construct diverse test formulas for SMT solvers by fusing pairs of equisatisfiable formulas. It effectively tackles both challenges of test input and oracle generation by constructing equisatisfiable formulas as the seed formulas via the concept of fusion and inversion functions. We have designed and engineered the practical bug detection tool YinYang to stress-test SMT solvers. Within four months of extensive testing, we have used YinYang to find 45 confirmed/fixed, unique bugs in Z3 and CVC4, 29 of which are the critical soundness bugs. Our effort is well-appreciated by the SMT solver developers — it is the largest, most successful testing campaign against SMT solvers, leading to drastically more bugs than any previous approaches.

For future work, within the context of SMT solvers, it would be interesting to explore the automatic generation of fusion and inversion functions, and to construct effective customized reducers for SMT formulas. Because solver clients often interact with the solvers via their provided APIs, it would also be interesting to adapt *Semantic Fusion* to test the solver APIs. Beyond validating SMT solvers, *Semantic Fusion* is a general technique and may be adapted to other domains; examples include the testing of database engines, compilers, and numerical solvers. This work opens up this exciting new direction via a successful application of *Semantic Fusion* to the testing of SMT solvers.

Acknowledgements

We thank our shepherd, Isil Dillig, and the anonymous PLDI reviewers for their valuable feedback. We are also grateful to Tim King for a corrected proof of Proposition 1. Our special thanks goes to the Z3 and CVC4 developers, especially Nikolaj Bjørner, Andrew Reynolds, and Andres Nötzli, for useful information and addressing our bug reports. Chengyu Zhang was partially supported by China Scholarship Council, and NSFC Projects No. 61632005 and No. 61532019.

References

[1] 2019. Using the GNU Compiler Collection (GCC): Gcov. Retrieved 2019-10-30 from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV*. 171–177.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2019. The Satisfiability Modulo Theories Library (SMT-LIB). Retrieved 2019-10-30 from www.SMT-LIB.org

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

[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A fuzzer for string solvers. In *CAV*. 45–51.

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

[7] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *SMT*. 1–5.

[8] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *SAT*. 44–57.

[9] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. https://doi.org/10.3929/ethz-b-000375243

[10] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *ASE*. 768–778.

[11] Cristian Cadar and Alastair Donaldson. 2016. Analysing the Program Analyser. In *ICSE*. 765–768.

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

[13] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report.

[14] The International SMT Competition. 2019. SMT-COMP. Retrieved 2019-10-30 from https://smt-comp.github.io/2019/index.html

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

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

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

[18] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. In *OOPSLA*.

[19] Vijay Ganesh, Dmitry Blotsky, Federico Mora, Ifaz Kabir, Murphy Berzish, and Yunhui Zheng. 2019. StringFuzz. Retrieved 2019-10-30 from http://stringfuzz.dmitryblotsky.com/

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

[21] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *ASE*. 590–600.

[22] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *ISSTA*. 239–250.

[23] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. 216–226.

[24] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*. 386–399.

[25] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. In *PLDI*. 65–76.

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

[27] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools keep their promises?. In *ESEC/FSE*. 331–341.

[28] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *ISSTA*. 176–186.

[29] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *PLDI*. 335–346.

[30] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *TSE* (2016), 805–824.

[31] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. EECS Dept., UC Berkeley.

[32] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *OOPSLA*. 849–863.

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

[34] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective dynamic detection of alias analysis errors. In *ESEC/FSE*. 279–289.

[35] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *ESEC/FSE*. 763–773.

[36] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI*. 347–361.