Value Mutation Testing for SMT Solvers

Dylan Wolff

Master's Thesis
Computer Science Department
ETH Zürich
October 2020

Dylan Wolff: *Value Mutation Testing for SMT Solvers*

Supervisors:
Dominik Winterer,
Manuel Rigger,
Zhendong Su

Location:
Zurich, CH

Value Mutation Testing for SMT Solvers

Dylan Wolff
ETH Zurich
wolffd@ethz.ch

***Abstract*—We investigate a subset of grammar-based mutational test case generation, *Value Mutation Testing*, as a methodology for finding bugs in SMT solvers. The core idea behind value mutation testing is to change the values that various constants take within a seed formula to produce a new formula with different behavior but the same valid syntactic structure. We then use differential testing to evaluate various solvers against these newly generated formulas. We further develop a novel methodology, *Boolean Sub-Expression Coercion*, to choose constant mutations with higher potential to trigger bugs. Here, we use constraints derived from the original formula as inputs in a separate call to an SMT solver. We then take new constant values from the resulting model of this call. To evaluate our approach, we implement ValFuzz, a value mutation fuzzer that uses Boolean Sub-Expression Coercion and differential testing to test SMT Solvers. With ValFuzz, we uncovered bugs in two state-of-the-art SMT solvers, filing 11 new bug reports on their respective issue trackers. We further demonstrate with ValFuzz, using a set of known bugs, that Boolean Sub-Expression Coercion is an effective method for picking value mutations compared to a randomized baseline.**

I. Introduction

Satisfiability Modulo Theory (SMT) solvers evaluate the satisfiability of first-order logic formulas that also contain functions from various external theories (e.g. the theory of linear arithmetic, unicode strings etc.). Such solvers are key components in a variety of tools. For many domains in which these tools are applied, such as program verification [1, 15, 19] or program synthesis [18, 28], correctness is crucial. AdaCore's SPARK-Pro, for example, is one such commercial product that utilizes underlying SMT solvers for verification. It has been used to create software for tactical military operations[1] and air traffic control systems.[2] Soundness errors in the underlying solvers for these products could result in incorrect code making it to a production environment with serious potential consequences. Thus, finding and fixing bugs in SMT solvers is also of vital importance. This is especially true for soundness bugs, which cause the solver to silently output an incorrect answer without a corresponding error message. Until recently, SMT solvers were generally considered to be robust and well-tested, though recent works [20, 31, 32] suggest that they are more commonly affected by serious bugs than previously thought.

We present one example of a formula that triggers such a soundness bug in Figure 1a. With this formula as input, Z3, an SMT solver, incorrectly reports that the constraints are satisfiable [12]. However, in this formula, the variable b is constrained to be equal to the integer value of a string composed of up to two non-negative digits, but is also asserted to be less than zero. This is a logical contradiction, and thus the formula should be unsatisfiable. However, if we transform the formula in Figure 1a by changing the highlighted constant from `0` to `11` (seen in Figure 1b), Z3 correctly reports this new formula to be satisfiable. Thus mutating constants in a given formula can hide buggy behavior in a solver. Conversely, mutating constant values in a non-buggy seed formula such as in Figure 1b can result in a new formula that triggers a bug (as in Figure 1a). This latter observation serves as our primary motivation for exploring Value Mutation Testing as a bug-finding technique for SMT solvers; by changing the constants within a given seed formula, we can rapidly generate a large set of new, syntactically valid input formulas that may trigger bugs.

Fig. 1: Two Similar SMT-LIB Formulas

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

(a) A bug triggering formula for Z3

```
(declare-fun a () String)
(declare-fun b () String)
(assert (= (str.++
    (str.substr "1" 0 (str.len a)) "0") b))
(assert (< (str.to.int b) 11 ))
(check-sat)
```

(b) A non-triggering mutation of the formula

Definition I.1. *We define a **Value Mutation** for a given seed formula ϕ with n constant values $c_1 \dots c_n$ as a transformation replacing each constant c_k in ϕ with a constant value c'_k of the same sort such that $\phi \neq \phi[c'_1 \dots c'_n / c_1 \dots c_n]$. We consider the new formula $\phi[c'_1 \dots c'_n / c_1 \dots c_n]$ to be a value mutant of ϕ.*

Finding a bug-inducing test case via naive random value mutations of the formula in Figure 1b, however, is highly unlikely. For this particular bug to trigger, the string constants in the formula must be digits, the first numerical constant must be a value such that it creates a valid sub-string of the first

[1]https://www.adacore.com/customers/cross-domain-guard
[2]https://www.adacore.com/customers/uks-next-generation-atc-system

string constant, and the last numerical constant needs to be zero. Meeting these constraints through the random selection of strings and integers, for example, is highly improbable. Indeed, during our evaluation, we were unable to reproduce this bug via random value mutations in practice. This observation motivates our strategy for value mutation: using constraints derived from the structure of the formula itself to allow an SMT solver to pick constant values that are more likely to produce interesting, divergent behavior in the SMT Solver under test. Our basic approach for these additional constraints, Boolean Sub-Expression Coercion, pushes the boolean sub-expressions within the original formula to take on different truth values with each generated test case. For example, we might attempt to find a new constant to replace `0` such that the boolean expression `(< (str.to.int b) 0)` evaluates to $false$. Indeed, with Boolean Sub-Expression Coercion and additional refinements discussed in Section V, we are able to find a bug in practice from the seed formula in Figure 1b.

To evaluate our approach of Value Mutation Testing using Boolean Sub-Expression Coercion, we implemented ValFuzz, a differential value mutation fuzzer for SMT-LIB formulas and their solvers [21]. Here differential means that we test different versions and configurations of solvers against each other on each test case, allowing us to find discrepancies between them that could be soundness bugs. We find that Value Mutation Testing is capable of discovering new bugs in two state-of-the-art SMT solvers, Z3 and CVC4, identifying a soundness issue in Z3 and eleven other bugs between both solvers [3, 12].

We additionally perform an analysis of ValFuzz on a set of known bugs from previous versions of Z3 and CVC4 and use these results to determine the effectiveness of Boolean Sub-Expression Coercion relative to various refinements of that approach and a randomized value mutation fuzzer baseline. Our experiments show that Boolean Sub-Expression Coercion was significantly more efficient at finding bugs in this data-set when compared with the randomized baseline. Furthermore, we saw that several modifications to this core technique improve ValFuzz's ability to discover these bugs. We believe that these results show that value mutation testing and Boolean Sub-Expression Coercion have potential as a novel testing technique for SMT solvers.

In summary, we provide the following contributions:

1) We define a technique for finding bugs in SMT solvers, Value Mutation Testing
2) We develop a novel methodology, Boolean Sub-Expression Coercion, to effectively and efficiently conduct value mutations on SMT formulas in practice
3) We implement Value Mutation Testing and Boolean Sub-Expression Coercion in a new tool, ValFuzz
4) We evaluate ValFuzz on state-of-the-art SMT solvers, identifying and reporting 11 new bugs in Z3 and CVC4

II. Background

In the following section, we present background information on the SMT-LIB format, the kinds of bugs we consider in SMT solvers, and some relevant context of prior work in dynamic testing of SMT solvers and other similar programs.

A. SMT-LIB

Most solvers accept inputs that adhere to the SMT-LIB [2] specification.[3] SMT-LIB formulas are composed primarily of declarations, definitions, and assertion statements, typically followed by a check for satisfiability. For example, Figure 1b declares two variables `a` and `b`, then two constraints with **`assert`** statements, and, finally, checks these constraints for satisfiability. In SMT-LIB formulas **`declare-fun`** statements referring to functions without arguments denote variable declarations. Each **`assert`** statement specifies constraints on the formula in the form of a boolean expression. This boolean expression may, in turn, have further sub-expressions of various sorts such as strings, integers, reals, booleans, and bit-vectors. These sorts may even be user-defined. Such expressions can also contain both variables and constant values. Figure 1b, for example, has four such constant values: two string constants, `"1"` and `"0"` and two identical numerical constants, both with a value of `0`. The precise ranges of valid constant values for each sort is, again, outlined in the SMT-LIB language. Operators for various expressions are written in prefix form, and precedence is indicated by parentheses.

The **`check-sat`** statement instructs the solver to determine whether an assignment of values to each variable exists such that each assertion evaluates to $true$. If such an assignment exists, then the formula as a whole is reported to be "sat" (satisfiable). If it is impossible to find such an assignment, the solver instead returns "unsat" (unsatisfiable), and if the solver cannot determine whether the formula is "sat" or "unsat", it will return "unknown". This check for satisfiability can also be followed by a `get-model` statement, which, if the formula is satisfiable, causes the solver to return a model that includes the satisfying values for each variable in addition to a "sat" result.

B. Bugs in SMT Solvers

The bugs in SMT solvers we consider can be categorized broadly into three groups: soundness bugs, invalid model bugs, and crash bugs. We also recognize assertion violations (from versions of the solvers compiled with such checks) as bugs, though they often manifest themselves as one of the other classes of issues with these assertions turned off. Of all classes of bugs in these solvers, soundness issues are the most serious because they involve the solver silently emitting an incorrect answer rather than aborting, making such bugs difficult to detect. Invalid model bugs occur when the `get-model` query returns values that do not represent a satisfiable assignment (although such an assignment does exist). Finally, crashing bugs are typically characterized by segmentation faults, and we consider them to be relatively less serious as the faulty behavior is clearly recognizable to an end-user or tool.

[3]The full language (and individual theory) descriptions and specifications are available on the SMT-LIB website [4]

C. Dynamic Testing

1) Structured Inputs: Testing programs with highly structured inputs is an open problem in the field of dynamic testing [7]. A common solution to create tests that pass the input checks of their target programs is to use a grammar-based generational approach to test-case generation [8]. Here, test cases are generated from scratch by a randomized process in conformity with a grammar specified for the input space. These approaches can accurately capture the syntax of the inputs, but do not necessarily also generate inputs with interesting behavior at runtime. Another approach is to mutate valid inputs in a syntax preserving manner. This technique starts with a broad base of diverse seeds rather than needing to generate semantically interesting test cases completely from scratch [33]. Previous work from this latter category by Winterer et al. [31] has demonstrated that type-preserving mutation of operators in SMT formulas can uncover many new bugs, including high-impact soundness issues.

2) Coverage and Symbolic Execution: First discussed in 1963, code and branch coverage have long been measures of the effectiveness of testing frameworks [22]. More recently, symbolic execution has been used as an effective means of increasing code coverage for dynamic testing, specifically fuzzing, both alone and in conjunction with other techniques [16, 29]. However, Winterer et al. notably observe that for SMT solvers, specifically, code coverage of the system under test does not appear to adequately explain the effectiveness of their approach. Thus it is an open question if there are other similar heuristics for SMT solvers that provide more accurate feedback for dynamic tests.

III. Approach

At a high level, our approach involves changing the values of constants within a given seed formula to produce a new formula that we then use as a test case for SMT solvers. We call this Value Mutation Testing. To pick the new values for each constant, we use a novel technique Boolean Sub-Expression Coercion. For Boolean Sub-Expression Coercion, we leverage constraints derived from the boolean sub-expressions of the seed formula, along with an additional call to an SMT solver to obtain our value mutations. By coercing the individual sub-expressions to be $true$ or $false$, and allowing the model from this intermediate SMT solver call to dictate which constant values will be used for a given formula, we explore values that are more likely trigger interesting behavior in the solver under test. In the following section we describe both Value Mutation Testing (Section III-A) and Boolean Sub-Expression Coercion (Section III-B) in detail.

A. Value Mutation Testing

Algorithm 1 illustrates our basic approach to Value Mutation Testing. As an input, the procedure expects a set of initial seed files, a set of SMT solvers under test, and an execution limit ℓ. To produce a new formula from a given initial seed ϕ, we begin by extracting references to each of the constants $c_1 \dots c_n$ within ϕ. We consider these references to be holes in the formula that we can replace with new values or variables. We denote these holes as $\square_1 \dots \square_n$. Thus, these holes and the original formula together give us a representation that we call the *skeleton* of ϕ.

Definition III.1. *We define the* ***Skeleton*** *of a given formula ϕ containing constant values $c_1 \dots c_n$ to be $\phi[\square_1 \dots \square_n / c_1 \dots c_n]$, where $\square_1 \dots \square_n$ represent holes in the new formula.*

We present an example skeleton for a simple SMT-LIB formula in Figure 2b.

From this representation of the skeleton of ϕ, we then pick values $c'_1 \dots c'_n$ to replace each constant hole. In order to pass basic validation checks within the SMT solvers under test, each new constant c'_k must be appropriately typed. There are many potential methods for picking these new values, but we elide this portion of the algorithm here to provide a more clear summary of the high-level approach. We discuss our specific strategy for choosing value mutations in the following subsection. Once new values have been picked for the holes, we substitute them in to the original formula, resulting in a new formula, $\phi[c'_1 \dots c'_n / c_1 \dots c_n]$, which we abbreviate ϕ^{sub}. Finally, we validate our solvers under test against ϕ^{sub}, recording any bugs encountered during this differential test.

Algorithm 1 Value Mutation Testing

```
1: procedure ValueMutationFuzz(Seeds, Solvers, ℓ)
2:     bugs = ∅
3:     for φ in Seeds do
4:         □_1 ... □_n ← ExtractHoles(φ)
5:         while c'_1 ... c'_n ← PickVals(□_1 ... □_n, φ, ℓ) do
6:             φ^sub ← Substitute(c'_1 ... c'_n, □_1 ... □_n, φ)
7:             if ¬Validate(φ^sub, Solvers) then
8:                 bugs ← bugs ⋃ {φ^sub}
       return bugs
```

B. Picking Values With Boolean Sub-Expression Coercion

1) Intuition: Determining how to pick meaningful values —those likely to trigger different behavior in SMT solvers— is an important consideration for value mutation testing. Many assignments to the holes of a formula will result in test cases that are not solved differently in practice. For example, changing a constant from `7` to `6` in Figure 2a, resulting in Figure 2d, is unlikely to exercise different behavior within the solver. The formula in Figure 2d is still satisfiable; the truth values of the individual sub-expressions do not change, and, indeed, Z3 outputs the same model for both formulas ($x = 0$). A naive approach of picking values from the uniform random distribution of possible constants, for instance, could result in choosing many such similar mutations —not an efficient way to explore the space of possible assignments to each hole.

In the context of conventional programming languages, branch coverage is well known to be positively correlated with fault discovery rate [10, 22]. As a result, symbolic execution has been a mainstay of fuzzing techniques because it allows

Fig. 2: Transformations of a Simple SMT-LIB Formula

```
(declare-fun x () Int)
(assert (or (< x 7) (> x -1)))
(check-sat)
```

(a) A simple SMT-LIB formula

```
(declare-fun x () Int)
(assert (or (< x □1) (> x □2)))
(check-sat)
```

(b) The skeleton of a simple SMT-LIB formula

```
(declare-fun x () Int)

; Fresh variables representing holes
(declare-fun h1 () Int)
(declare-fun h2 () Int)

(assert (or (< x h1) (> x h2)))
(check-sat)
```

(c) The SMT-LIB skeleton of a simple SMT-LIB formula

```
(declare-fun x () Int)
(assert (or (< x 6) (> x -1)))
(check-sat)
```

(d) Mutating a constant in a simple SMT-LIB formula

```
(declare-fun x () Int)
(assert (or (< x 6) (> x 7)))
(check-sat)
```

(e) Mutating constants in a simple SMT-LIB formula

fuzzers to pick meaningful values that efficiently explore the input space in terms of branch coverage [17] [23] [29]. The intuition for our approach for picking value mutations is to consider the boolean sub-expressions of a formula to be analogous to the branching statements of a typical program. Thus we hypothesize that we can use the values taken by the various boolean sub-expressions of a satisfiable formula as a heuristic for identifying meaningfully different mutations; we consider two formulas that share a skeleton to produce meaningfully different executions if their solutions under a solver S have different boolean sub-expression saturation coverages.

Definition III.2. *The **Boolean Sub-Expression Saturation Coverage** for a solution of a satisfiable ϕ under a solver S is the set of equality relations between the boolean sub-expressions $\beta_1^{skel} \dots \beta_n^{skel}$ of ϕ^{skel} and the truth values of their corresponding sub-expressions in the model of ϕ under S.*[4] *In essence*

$$S(\phi) \Rightarrow (\beta_1^{\phi} = \tau_1 \wedge \dots \wedge \beta_n^{\phi} = \tau_n)$$

where $\tau_k \in \{\top \mid \bot\}$ has coverage of

$$\{(\beta_1^{skel} = \tau_1), \dots, (\beta_n^{skel} = \tau_n)\}$$

[4]This definition can be extended to include unsatisfiable formulas, where we consider different minimum unsatisfiable cores to be divergent behavior, but, for the purposes of this paper, we will focus on the satisfiable case.

We consider the saturation coverage of multiple formulas to be the set-union of their individual saturation coverages. This idea of saturation is used commonly for branch coverage in other fuzzing contexts [17].

For example, the formula in Figure 2e produces a model ($x = 8$) under Z3 in which one of the sub-expressions is $true$ and the other is $false$, whereas in the previous cases, Z3 output a model ($x = 0$) for which both sub-expressions evaluate to $true$:

Model for Figure 2a ($x = 0$):

$$(x = 0) \Rightarrow ((x < 7) = true) \wedge ((x > -1) = true)$$

Model for Figure 2d ($x = 0$):

$$(x = 0) \Rightarrow ((x < 6) = true) \wedge ((x > -1) = true)$$

Model for Figure 2e ($x = 8$):

$$(x = 8) \Rightarrow ((x < 6) = \textit{false}) \wedge ((x > 7) = true)$$

Thus, we consider the formula in Figure 2e to have different boolean saturation coverage under Z3 relative to Figure 2a and 2d. Similarly, we do not consider Figure 2a and 2d to have different coverage in relation to each other by this metric.

2) Boolean Sub-Expression Coercion: This subsection outlines our approach to efficiently achieve higher boolean sub-expression saturation coverage for value mutation: *Boolean Sub-Expression Coercion*. At a high level, for a seed ϕ, this technique consists of constructing an SMT formula that asserts a boolean sub-expression from ϕ^{skel} to be a truth value that has not yet been covered, and then letting an SMT solver pick constant values that are likely to cover that branch in the final test formula. This approach is exemplified in Figure 3.

Definition III.3. *We define the **SMT-LIB Skeleton** of a given formula, ϕ, as another SMT-LIB formula identical to ϕ, but with each constant value replaced by a fresh, initialized SMT-LIB variable representing a hole. This is similar to our conceptual definition of a skeleton, but is a valid formula that can be passed directly to an SMT Solver. We abbreviate the SMT-LIB skeleton of ϕ as ϕ^{skel}. We present a formula and its SMT-LIB skeleton in Figure 2a and 2c respectively.*

In Step 1 of Boolean Sub-Expression Coercion, we first enumerate and replace all of the constant values $c_1 ... c_n$ within the formula with fresh global variables $h_1 ... h_k$, creating ϕ^{skel}.

Definition III.4. *We define the **Boolean Monitor Variables** $m_1 \dots m_n$ of a formula ϕ to be a series of variables that are asserted to have the same value as some unique constraint derived from ϕ.*

Definition III.5. *Subsequently, we define the **Meta-Formula** ϕ^{meta} for a seed ϕ to be a formula containing the declarations and equivalence assertions for a series of boolean monitor variables $m_1 \dots m_n$ of ϕ^{skel} and their corresponding constraints, as well as any sort and function declarations from ϕ^{skel} necessary for ϕ^{meta} to be a valid formula.*

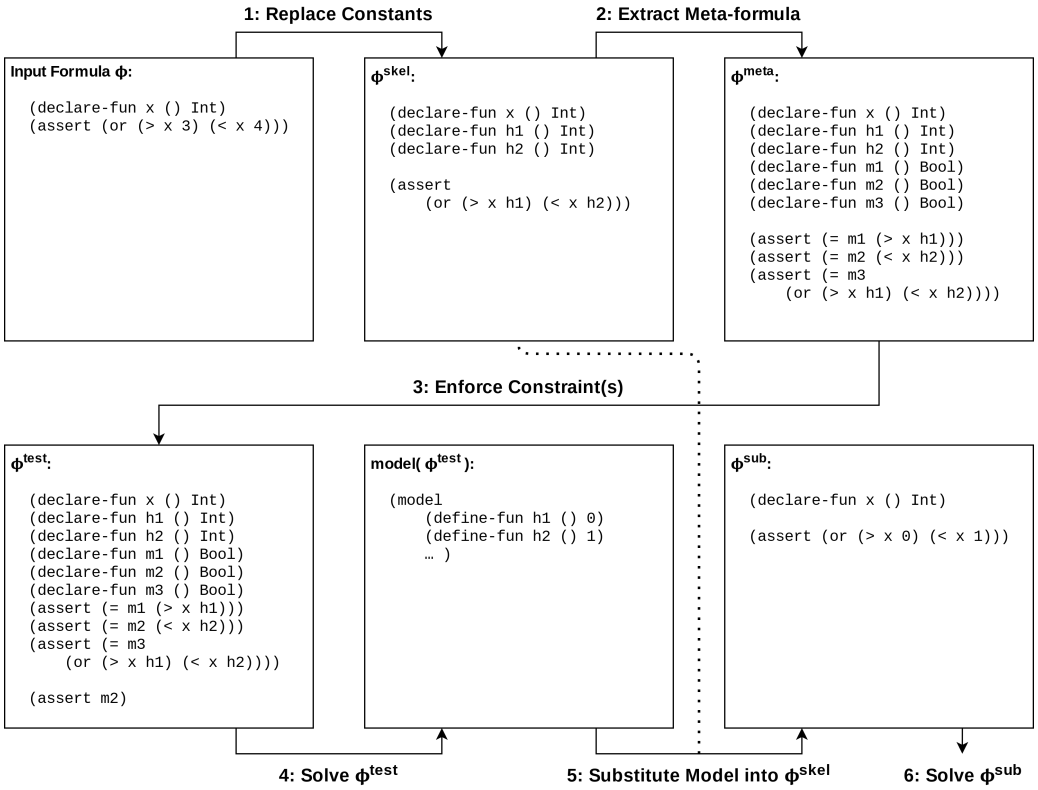


Fig. 3: A Single Iteration of Boolean Sub-Expression Coercion

In Step 2, from ϕ^{skel}, we construct a meta-formula with boolean monitor variables for each boolean sub-expression within ϕ^{skel}. In the example illustrated in Figure 3, there are three such expressions in ϕ^{skel}, corresponding to both branches of the `or` expression and the `or` expression itself. Thus, in ϕ^{meta} there are three monitor variables, each asserted to be equivalent to their respective sub-expressions. By construction, ϕ^{meta} is trivially satisfiable.

In Step 3 of Boolean Sub-Expression Coercion, we append an assertion that one of the boolean monitors of ϕ^{meta} is either $true$ or $false$, producing ϕ^{test}.[5] Note that there are many different ϕ^{test} that are produced by a single ϕ^{meta}; each iteration of the fuzzer generates one such unique ϕ^{test}.

In Step 4 of Boolean Sub-Expression Coercion, once a ϕ^{test} has been generated, we then pass it to an SMT solver. If the SMT solver finds that the formula is satisfiable, we can use values from the model to substitute in new values for the constants holes of ϕ^{skel} in Step 5. We call the formula with these substitutions ϕ^{sub}. Once we have a ϕ^{sub}, we can pass it directly to the solver(s) under test for differential testing, Step 6 of Boolean Sub-Expression Coercion.[6]

[5]For our basic approach to Boolean Sub-Expression Coercion, we enforce precisely one such monitor variable at a time. This ensures that the resulting ϕ^{test} is typically satisfiable. Simultaneously enforcing multiple constraints is discussed in Section V-E.

[6]Note that enforcing constraints on the meta-formula does not guarantee that they will hold for the final formula in Step 6 of Figure 3. This is discussed further in Section V-G.

If the solver finds that ϕ^{test} is unsatisfiable, or cannot find any solutions to the formula, then we return to the previous step (Step 4 in Figure 3) to create a new ϕ^{test}.

Algorithm 2 Boolean Sub-Expression Coercion

1: **procedure** PICKVALUES($holes$, ϕ, ℓ)
2: $\quad\phi^{skel} \leftarrow$ REPLACECONSTANTS$(holes, \phi)$
3: $\quad\phi^{meta} \leftarrow$ EXTRACTMETA(ϕ^{skel})
4: $\quad$**while** $\phi^{test} \leftarrow$ ENFORCECONSTRAINT$(\phi^{meta},\ \ell)$ **do**
5: $\quad\quad$**if** $model \leftarrow$ SOLVE(ϕ^{test}) **then return** $model$
$\quad$**return** None

We present a sketch of the algorithm for Boolean Sub-Expression Coercion as described above in Algorithm 2. Here we see that, for a given seed formula ϕ, we first replace the constants with fresh variables, creating the SMT-LIB skeleton ϕ^{skel}. From this SMT-LIB skeleton, we create the meta-formula and then enforce constraints up to saturation or an execution limit ℓ. For each resulting model, we then return that model for substitution as new constant values in the overarching value mutation procedure from Algorithm 1.

IV. IMPLEMENTATION

We realize our approach to value mutation testing in ValFuzz, a Rust program for testing SMT Solvers. The following section describes the relevant implementation details of ValFuzz.

A. Local Variables

Many SMT-LIB formulas contain syntactic constructs that prevent sub-expressions from being syntactically valid SMT-formulas when extracted to the meta-formula. Specifically, **let** expressions and quantifiers both introduce what amount to locally scoped variables. Naive extraction of sub-expressions containing these constructs as described in Section III is syntactically invalid because the variables are not initialized outside of the context of the enclosing expression that introduced them. For example, extracting the sub-expression `(< x 4)` from the following:

```
(assert (let (x (8))(and (< x 3) (< x 4))))
```

as

```
(declare-fun m1 () Bool)
(assert (= m1 (< x 4))))
```

...is syntactically invalid because the variable `x` has not been initialized in the extracted formula.

ValFuzz handles **let** expressions primarily by instantiating a fresh global variable of the same sort for each variable introduced by the **let** expression. So in the previous example, the resulting formula would be:

```
(declare-fun m1 () Bool)
(declare-fun l1 () Int)
(assert (= m1 (< l1 4))))
```

Here the fresh global variable `l1` replaces the local variable `x`, resulting in an equivalent formula in which all variables are properly declared.

The core implementation of ValFuzz does not handle quantifiers, and thus it is restricted to operating on quantifier-free logics. Modifications to accommodate quantifiers requires additional considerations discussed in Section V-C

B. Using SMT Solvers to Test SMT Solvers

Because ValFuzz uses the very kind of program it is testing, SMT solvers, to generate test cases, it is possible that a solver could have a bug, or even documented behavior, that manifests itself during the test-case generation phase, hiding bugs that would otherwise be discoverable with Value Mutation Testing. To avoid this possibility, ValFuzz uses multiple solvers and versions in Step 4 in Figure 3 in case of errors or failures. Furthermore, using multiple solvers allows us to differentially test the solvers against ϕ^{test} in addition to the final substituted formula ϕ^{sub}. Indeed, this intermediate step was actually responsible for finding some of the bugs we reported in Z3 and CVC4. For example, the crash bug shown in Figure 4a and 4b is not triggered by the constants originally present in the formula. When these constants are replaced with free variables and are part of an enforcement in ϕ^{test} or a validation check of ϕ^{skel}, however, they trigger an assertion violation in Z3.

C. Flags

To generate a diverse model with each iteration, ValFuzz uses various flags to randomize the outputs of the solvers in Step 4 of Figure 3. These command-line options are listed in Appendix B. ValFuzz also differentially tests across multiple solver configurations in addition to across solvers. These options are, likewise, listed in Appendix B.

Fig. 4: Two Similar SMT-LIB Formulas

```
(declare-const c (_ FloatingPoint 11 53))
(assert
    (= c ((_ to_fp 11 53) roundTowardZero 0.5 0)))
(check-sat-using smt)
```

(a) Non-triggering formula

```
(declare-const a Int)
(declare-const b Real)
(declare-const c (_ FloatingPoint 11 53))
(assert
    (= c ((_ to_fp 11 53) roundTowardZero b a)))
(check-sat-using smt)
```

(b) Replacing constants with variables triggers a bug

V. Modifications and Refinements

In addition to implementing Boolean Sub-Expression Coercion in ValFuzz, we also made several modifications to this core technique. The following section outlines these additions to ValFuzz.

A. Abstract Domains

Abstract interpretation has been a mainstay of static analysis techniques for decades [11]. In abstract interpretation, the values composing the state of a given program are represented as *Abstract Domains*. One example of such a domain would be to use the intervals of $\mathbb{Z}^-$, 0 and $\mathbb{Z}^+$ in place of integer values.

Our intuition is that, by abstracting over the domain of values that variables can take within a formula, we can push the solvers to efficiently explore this space of possible values. In essence, we hypothesize that formulas in which variables

TABLE I: Abstract Domains For An Expression "s"

Sort	Assertion
Bool	$s = true$
	$s = false$
Int/Real	$s = 0$
	$s < 0$
	$s > 0$
String	$s =$ ""
	$s \neq$ ""
Floating Point	$s = NaN$
	$s = -\infty$
	$-\infty < s < -0$
	$s = -0$
	$s = +0$
	$+0 < s < +\infty$
	$s = +\infty$

take on values within different intervals of our hand-picked domains are more likely to produce divergent behavior than values within the same intervals.

Thus, for this modification, we added abstract domain constraints to ϕ^{meta} for each variable in ϕ. The abstract domain constraints we used are illustrated in Table I. For example, if we were to generate such constraints for the variable x in the formula from Figure 3, where x is an integer, the following boolean monitors would be added to ϕ^{meta}:

```
(assert (= m4 (= x 0)))
(assert (= m5 (< x 0)))
(assert (= m6 (= x 0)))
```

These monitors are then enforced to saturation along with the other monitor variables according to the basic approach in Step 3, after which point the value mutation process continues as in the basic approach of Boolean Sub-Expression Coercion.

In addition to using abstract domain constraints for the variables in each seed formula, we explored generating these constraints for each sub-expression within that formula as well. With this refinement enabled, an expression such as `(< x (+ 1 y))` would produce the constraints:

```
(assert (= m1 (= (+ 1 y) 0)))
(assert (= m2 (< (+ 1 y) 0)))
(assert (= m3 (= (+ 1 y) 0)))
```

... in addition to any domain constraints on the variables `x` and `y`.

B. Leaf Expression Optimization

We define the *Leaf Expressions* of a formula to be the compound expressions (i.e. not a single variable or constant) that themselves contain no compound sub-expressions of the same sort. For example, the **`and`** expression below:

```
(and (< x 2) (> x 3))
```

has two leaf expressions: `(< x 2)` and `(> x 3)`. The encompassing `and` expression itself, however, is not a leaf expression, because it contains two compound sub-expressions that are also booleans.

For a given formula ϕ, the leaf sub-expressions are often sufficient to describe the behavior of that formula. In the previous example, the behavior of the expression in question can be fully described by the sub-expressions `(< x 2)` and `(> x 3)`: by exercising all possible combinations of truth assignments to these two sub-expressions, we have also necessarily exercised all possible values of the encompassing `and` expression. By only taking the leaf expressions of the original formula to construct ϕ^{meta}, we can reduce the number of boolean monitor variables, often without missing any of the behavior of the non-leaf sub-expressions. As a result, we remove redundancy in the search space of possible ϕ^{test}'s and, ideally, improve the efficiency of Value Mutation testing via Boolean Sub-Expression Coercion.

C. Quantifier Handling

Quantifiers present a similar problem to **`let`** expressions, as discussed in Section IV-A, in that they introduce localized variables to their sub-expressions. Thus, to extract these sub-expressions to be used in the constraints of ϕ^{meta}, we need to take special care that quantified variables within them are initialized properly. Without such modifications, ValFuzz cannot handle quantified formulas without producing syntactically invalid formulas, which precludes fuzzing a large swath of seed formulas available that utilized quantified logics. In this section, we discuss two methods that ValFuzz uses to generate tests for these formulas.

Positive (non-negated) existential qualifiers can be removed from a formula via *skolemization*, in which the quantified variables are each replaced by a fresh global variable [26]. This process results in an equisatisfiable formula. An example of skolemization is shown below for the following simple formula:

```
(assert (exists (x Int) (> x 3))
```

... which, when skolemized, becomes:

```
(declare-fun q1 () Int)
(assert (> q1 3))}
```

For positive universal quantifiers, however, this transformation does not result in an equisatisfiable formula. This is because the transformation relaxes the universally qualified variable to be existentially quantified instead. Thus it is not immediately clear that universal quantifiers should be handled in this way.

Another possibility would be to copy the universal quantifier along with the corresponding sub-expression into our constraint. For example, extracting the boolean sub-expression `(> x 4)` from the following formula into a boolean sub-expression monitor statement:

```
(assert (forall (x Int) (or (> x 3) (> x 4))))
```

... we get:

```
(assert (= m4 (forall (x Int) (> x 4))))
```

This transformation has the drawback of producing monitor expressions that are unsatisfiable, and thus monitor variables that are always $false$, such as in the example above. Additionally, this transformation does not accurately capture the behavior of existential quantifiers in monitored expressions. This can be seen with a quick example using the same formula but with an existential quantifier:

```
(assert (exists (x Int) (or (> x 3) (> x 4))))
    →
(assert (= m4 (exists (x Int) (> x 4))))
(assert (= m5 (exists (x Int) (> x 3))))
```

Here the x in the first assertion for $m4$ can take on a different value than the x in the second assertion in $m5$, when, in the original formula, these x's are the same variable.

Unfortunately, in SMT-LIB formulas, sometimes quantifiers are negated. Thus, by the following identities, we cannot assume that an SMT-LIB **`exists`** statement can be skolemized –nor that a **`forall`** statement can be copied– without a significant simplification step:

$$\neg\forall x \Leftrightarrow \exists x \qquad \neg\exists x \Leftrightarrow \forall x$$

TABLE II: Quantifier Handling Methods Impact on Satisfiability for Formulas Containing Quantifiers (n=30)

	Sat Enforcements (± Std. Dev.)	Sat Substitutions (± Std. Dev.)
Files With ∀:		
Copy Quantifier	78.3 (± 0.5)%	30.7 (± 2.1)%
Skolemize Quantifier	93.5 (± 0.1)%	22.3 (± 0.8)%
Files With ∃:		
Copy Quantifier	87.6 (± 0.6)%	15.8 (± 4.6)%
Skolemize Quantifier	93.3 (± 0.3)%	28.6 (± 1.8)%

Furthermore, because we are concerned with creating constraints for enforcement as an intermediate step, it is not even clear that handling quantifiers in this way will result in better performance. In the case of copying quantifiers, it is well known that formulas containing universal quantifiers are more difficult for SMT solvers; even simpler logics like linear arithmetic are undecidable when quantifiers are added [25]. Furthermore, universal quantifiers themselves are a more strict constraint than existential quantifiers, as $\forall x.z \Rightarrow \exists x.z$ but $\exists x.z \not\Rightarrow \forall x.z$. This could result in more frequently unsatisfiable ϕ^{test}, which can be problematic, as our technique involves taking values from satisfiable results at this step.

To empirically understand how the quantifier handling technique affected fuzzing with value mutation, we took 2000 files from the SMT-LIB benchmark data-set and tested them with each strategy [4]. Of these files, 1000 contained universal quantifiers and the other 1000 contained existential quantifiers. We ran ValFuzz for 50 iterations per file on each set, with the corresponding type of quantifier either skolemized or copied. This setup was repeated 30 times, each using a different random seed for ValFuzz's internal random number generator. The results of this experiment are shown in Table II.

From this table, we can see that the satisfiability rate of each enforcement is significantly lower when universal quantifiers were copied, rather than skolemized as if they were existential quantifiers. This is a result of two contributing factors. Firstly, the rate of unsatisfiable ϕ^{test} was much lower when the quantifiers are skolemized, due to the strictly relaxed constraints, dropping by about 10%. Secondly, the rate of timeouts also decreased, as the formulas without universal quantifiers proved to be easier to solve. This rate fell by a smaller, but still significant, 1% (p-value < 0.0001). These two factors combined lead the skolemizing version of ValFuzz to generate roughly 10% more test cases on average.

One potential problem with the skolemizing relaxation is that a loss of equisatisfiability with respect to the original formula could result in fewer final substitutions that make sense in the context of the problem. In essence, that there will be far more unsatisfiable ϕ^{sub} because the constraints enforced at Step 3 of Figure 3 are weaker than the original formula. This imbalance is not necessarily a problem, but testing many trivially unsatisfiable formulas is also likely not a good usage of compute time. From Table II, we see that there was a significant increase in the unsatisfiable substitution rate of formulas containing universal quantifiers using skolemization. However, the opposite is true for formulas containing existential quantifiers. This mixed result, along with the significant drop in throughput when copying quantifiers, leads us to use skolemization of all quantifiers by default in our evaluation in Section VI.

D. Appending the Original Formula to the Meta-Formula

In our basic approach, the meta-formula consists of only expressions asserting the equivalence of monitor variables to their corresponding expressions. Of these monitor variables, we assert precisely one to be $true$ or $false$ in ϕ^{test}. This has the advantages of guaranteeing that ϕ^{meta} is satisfiable and typically producing ϕ^{test} that are satisfiable as well. The satisfiability of these formulas is important as the substitution at Step 5 in Figure III depends on the model produced by the solver at Step 4. It has the disadvantage, however, that these assertions lose the structure of the original problem. Thus, the model we get from solving at Step 4 may have constant values that are directly contradictory to the original formula, and thus can make the final ϕ^{sub} trivially unsatisfiable. For example, the enforcement in Step 3 of Figure 3 could result in the final test case containing `(assert (> x 0) (< x 0))`, which is clearly unsatisfiable. For these ϕ^{sub}, we hypothesize that the solver under test can be less likely to manifest a bug because it is able to deduce the formula is unsatisfiable relatively quickly, without exercising deeper code paths within various theory solvers. Furthermore, we hypothesize that an under-constrained ϕ^{test} results in less diverse solutions from the solver in Step 4 of Boolean Sub-Expression Coercion from Figure 3.

To add more structure to the meta-formula, we appended ϕ^{skel} and its negation, ϕ^{skel-}, to the meta-formula, creating two new meta-formulas: ϕ^{meta+} and ϕ^{meta-}. To create ϕ^{skel-} we simply add a negation to each assertion in the original formula. So, using the example from Figure 3, we append:

```
(assert (or (> x h1) (< x h2)))
```

...to ϕ^{meta} to get ϕ^{meta+} and likewise:

```
(assert (not (or (> x h1) (< x h2))))
```

...to get ϕ^{meta-}. Steps 3-6 can then proceed normally on both meta-formulas. By using two meta-formulas, we ensure that we will still typically get at least one "sat" result from solving ϕ^{test+} or ϕ^{test-} at Step 4; if ϕ^{skel} is unsatisfiable, ϕ^{meta-} will usually be "sat", and vice versa. In many cases, because the constants have been replaced with fresh variables representing holes, and thus have less restrictive constraints, both ϕ^{meta+} and ϕ^{meta-} are satisfiable, producing even more test cases. This, too, is desirable, as it still preserves the linearity of our saturation coverage metric by approximately doubling the number of test cases. In practice, because some test cases are duplicates, we found this increase to be approximately 26% on a set of 1000 files from the benchmark data-set (n=30) [4].

E. Multiple Constraint Enforcement

As mentioned in Section V-D, we observed that many of our ϕ^{test} were under-constrained relative to the original formula. For example, enforcing `(< x h2)` alone to be $true$ as in Figure III does not actually place any bounds on constant value for the hole $h2$; because x is also completely free, $h2$ can take on any value in the space of integers. One method for adding constraints is to append the original formula, as in Section V-D. Alternatively, we can enforce multiple constraints from ϕ^{meta} at once instead of only a single constraint, enforcing both $m1$ and $m2$ to be $true$ in ϕ^{test} of Figure III, for example. As more constraints are enforced simultaneously, the values constants can take are more restricted, hopefully producing more relevant and diverse results. However, as some constraints may contradict each other, adding more of these restrictions can also result in a much higher rate of unsatisfiable ϕ^{test}. In Section VI-B2b, we discuss selecting a number of constraints to enforce based on data from known bugs in Z3 and CVC4.

F. Relations Between Constants

To motivate our next modification to ValFuzz, we examine a bug triggering formula in Listing 1.

```
(declare-fun x () String)
(declare-fun y () String)
(assert (str.in_re
    (str.++ x "c" y)
    (re.++
        (re.* (str.to_re "c"))
        (re.union (str.to_re "a")
            (str.to_re "b")))))
(check-sat)
```

Listing 1: A Bug Inducing Formula for Z3

The bug triggered by the formula in Listing 1 manifests only when the first and second string constants are equal to each other and the third and fourth constants are not equal to each other. This example suggests that, in at least some cases, the relationship between constants is an important driver of SMT solver behavior. In an effort to capture these effects, we included the relations shown in Table III as additional constraints in the meta-formula. Thus, for the formula ϕ^{skel} in Figure 3, we would add the following constraint monitors to ϕ^{meta}:

```
(assert (m4 (= h1 h2)))
(assert (m5 (< h1 h2)))
(assert (m6 (> h1 h2)))
```

For n relations, a formula with k constants has $n * \binom{k}{2}$ possible combinations. Thus, for most moderately sized formulas, it is intractable to include all such constraints in the meta-formula. More importantly, the number of monitor variables introduced by these relations would dwarf the number of all other monitor variables, causing them to completely dominate testing runs, to the virtual exclusion of our basic approach and many other modifications. To offset this effect, we capped the number of constants used in these relations to be a relatively small subset of the whole.

TABLE III: Relational Assertions Between Holes $h1$ and $h2$

(Sort h1, Sort h2)	Assertions
(Bool, Bool)	$h1 = h2$
(String, String)	$h1 \neq h2$
(BitVec m n, BitVec m n)	
(Int/Real, Int/Real)	$h1 = h2$
(Floating Point e m, Floating Point e m)	$h1 \neq h2$
	$h1 > h2$
	$h1 < h2$
	$h1 \geq h2$
	$h1 \leq h2$

Which variables are included in the subset is determined by the random seed of the fuzzing run. We discuss finding an appropriate size for this subset in Section VI-B2b.

G. Enforcing Constraints at the Final Step

In our basic approach, there are no guarantees that the constraints we enforce in ϕ^{test} will actually hold when we solve ϕ^{sub}. For example solving ϕ^{test} from Figure III where $m2$ is enforced to be $true$, we might get a model giving:

$$h1 = 1,\ h2 = 0,\ x = 0$$

However, substituting the constants back into ϕ^{skel} and solving our new ϕ^{sub}, we might get a model in which $x = 2$. Clearly, in this case, our enforced constraint $m2$ does not hold, as the substituted values from the model makes the monitored sub-expression evaluate to $false$:

$$(x = 2) \Rightarrow ((x < h2) = m2 = false)$$

This discrepancy is not necessarily an issue, but it could result in many formulas with highly similar behavior, even when the constraints are varied in ϕ^{test}.

To determine if our core technique of Boolean Sub-Expression Coercion creates test cases with diverse boolean sub-expression coverages in practice, we added instrumentation to the generated test cases in the form of monitor variables from ϕ^{meta}. By propagating the monitor variables from ϕ^{meta} into ϕ^{sub}, we were able to track how frequently the monitor variable(s) enforced at Step 4 retained the same truth value in the model of Step 6. There are clearly at least two deficiencies with this experimental setup, however: it only works for satisfiable formulas and adding the monitor variables to ϕ^{sub} may, in fact, influence the outcome of the solver itself. Still, though imperfect, the experiment does give some insight into how frequently the enforcement from ϕ^{test} is held through the solver's result on ϕ^{sub}.

We ran this experiment 30 times on 1000 files from the benchmark data-set and found that the enforcement holds throughout the final step for 77.6% of satisfiable ϕ^{sub} with a standard deviation of 7.2% [4]. This is far better than the 50% rate we would expect if the enforcement had no-effect. While we do have this evidence that, at least most of the

time, our final formula is respecting constraints we added in the previous step, we also hypothesized that ensuring these constraints were always met could lead to better results.

One way to ensure that these constraints are enforced is to append assertions that enforce them to ϕ^{sub}. These extra assertions result in a final ϕ^{sub} that no longer shares a skeleton with ϕ. Still, we experimented with this modification, as retaining the original skeleton is not a necessary condition for testing. To do so, we include both the constraint monitors from ϕ^{meta} and the enforcement from ϕ^{test} in the new ϕ^{sub}. Again using the example enforcement from Figure 3, we would include the following in our new ϕ^{sub} in addition to the rest of the formula:

```
(declare-fun () m2)
(assert (= m2 (< x 1)))
(assert m2)
```

VI. Evaluation

We first evaluate Value Mutation Testing as a bug finding technique by determining if our implementation of this approach, ValFuzz, is capable of finding bugs in real-world target programs. This consisted of an extended fuzzing campaign of the Z3 and CVC4 solvers. We further collected a set of 90 known bugs from the CVC4 and Z3 issue trackers and utilized them for an additional evaluation of ValFuzz and its various refinements. These two evaluations: on newly-discovered and previously-known bugs are discussed in the proceeding sections.

One notable omission from our evaluation is that of code-coverage data. Previous work has shown code-coverage to be ineffective in explaining the success of their approaches, including specifically in the case of testing Z3 and CVC4 [31][32].

A. Newly Discovered Bugs

1) Fuzzing Setup: We intermittently evaluated ValFuzz in parallel with its development over the course of approximately three months, beginning at the end of May 2020. We performed fuzzing runs with various configurations primarily on an AMD Ryzen Threadripper 2990WX processor with 32 cores and 32GB RAM running Ubuntu 18.04 64-bit.

We utilized the SMT-LIB Benchmarks [4], in addition to the test suites for Z3 and CVC4 as seed files. We reduced and processed bugs for reporting using a combination of C-reduce and GNU Parallel [24, 30].

2) Reported Bugs: In total, ValFuzz uncovered 11 new bugs during our evaluation period. Its performance relative to other recent works in terms of bugs reported is summarized in Table IV. We would highlight, however, that the quantity of bugs reported is not sufficient to determine the effectiveness of a given technique. It does not necessarily, for example, give any indication of fuzzing efficiency. Furthermore, because other approaches were applied primarily prior to our evaluation with ValFuzz, it is possible that many of the bugs that could have been found by ValFuzz were already uncovered by these other approaches. Still, it is quite clear that YinYang, OpFuzz, and STORM ([20, 31, 32]), at the least, all are likely more effective than ValFuzz at finding bugs in SMT solvers due to the large disparity in reporting numbers. That being said, ValFuzz is competitive in this metric with other previous efforts such as StringFuzz, BanditFuzz, and the recent work by Bugariuet et al. [6, 9, 27].

TABLE IV: Comparison of Confirmed Bugs found by Value Mutation and Other Approaches [31]

	Bugs in Z3		Bugs in CVC4	
	soundness	all	soundness	all
StringFuzz	0	1	–	–
BanditFuzz	≥ 1	≥ 1	–	–
Bugariu et al.	3	5	0	0
YinYang	25	39	5	9
STORM	21	27	0	0
OpFuzz	114	452	16	180
ValFuzz	1	6	0	4

3) Reception: Our reports to developers in both projects seemed to be well received. A few comments from discussions on GitHub issues we filed include:

> *"Nice catch, I was able to make some major improvements to the regex algorithm and model construction with this one. Thanks for the report!"*
> (https://github.com/Z3Prover/z3/issues/4271)

> *"Good catch, recent regression"*
> (https://github.com/Z3Prover/z3/issues/4471)

> *"Thanks for the report, this is now fixed!"*
> (https://github.com/Z3Prover/z3/issues/4190)

To further understand the potential of value mutation as a bug finding technique, we proceed with a secondary analysis on the set of previously reported issues.

B. Previously Known Bugs

1) Experiment Setup: We hand-picked 90 total bugs from the issue trackers of CVC4 and Z3 that contained constant values. We attempted to group these issues into reproducing with as few different versions of Z3 and CVC4 as possible to facilitate automated testing, eventually reducing the set to 28 different versions of the solvers. Of the original 90 bugs, 19 were consistently evident as soundness issues via differential testing with no modifications, and 28 typically showed another class of bug. Because some files manifested multiple issues, we had 38 files total that showcased a bug. For each configuration of ValFuzz, we generated test cases using 50 iterations per-file on this data-set, with the reproducing solver version as the solver under test for each file. Iterations here correspond to Step 3 from Figure 3.

To evaluate the effectiveness of our approach to picking value mutations, we also implemented a completely random-

ized value mutation testing fuzzer to use as a baseline. It replaces constants by sampling from the uniform random distributions of their possible values.

We track both the quantity and class of each bug found by various configurations of ValFuzz and the randomized baseline, as well as an efficiency measure; specifically:

$$\frac{\text{Number of Non-Trivial Bugs}}{(\text{Solved Enforcements} + \text{Solved Substitutions})}$$

where Solved Enforcements and Solved Substitutions correspond to the number of times ValFuzz executed Steps 4 and 6 from Figure 3, respectively. We choose this as our efficiency metric because calls to the solvers dominate execution time for ValFuzz, and this number corresponds directly to the amount of differential solving operations during a fuzzing run. Absolute timing data is more volatile, especially in the multi-tenant environment in which these tests were performed. Furthermore, the absolute timing depends heavily on the number of configurations or solvers being tested. Thus, for certain configurations of ValFuzz, it may be more appropriate to use different weights for each group of calls to the solver. We believe, however, that our efficiency metric should give a productive baseline figure that can be taken into account when optimizing for a specific use case.

In this efficiency calculation, we also discount bugs that are trivial to find. To determine which bugs are trivial, we ran the completely randomized value mutation fuzzer 50 times with only a single iteration per file. We consider bugs that were found in each of the 50 runs to be trivial, in that the values chosen for constants within them do not seem to affect whether or not the bug is triggered. This amounted to subtracting 21 total bugs (including one soundness bug) from the values in the corresponding columns in Table V.

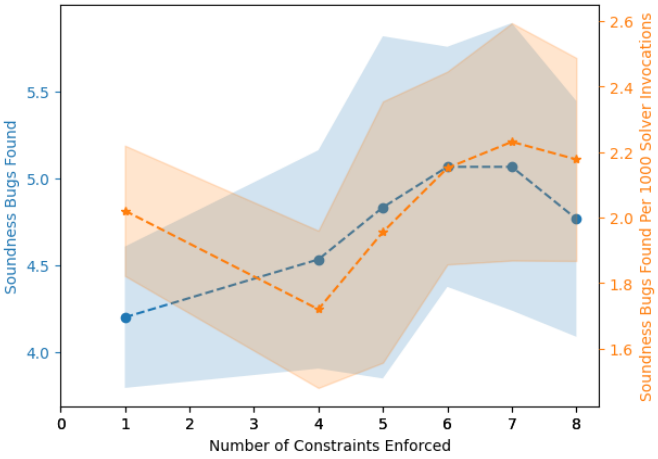
We tested each refinement individually and then selected two combinations of modifications that improved significantly (determined via Welch's t-test) from the basic approach in either the absolute number of soundness bugs found or the efficiency with which these soundness bugs were found. We also tested how the leaf optimization discussed in Section V-B affected our most effective blend of modifications.

2) Experiment Evaluation:

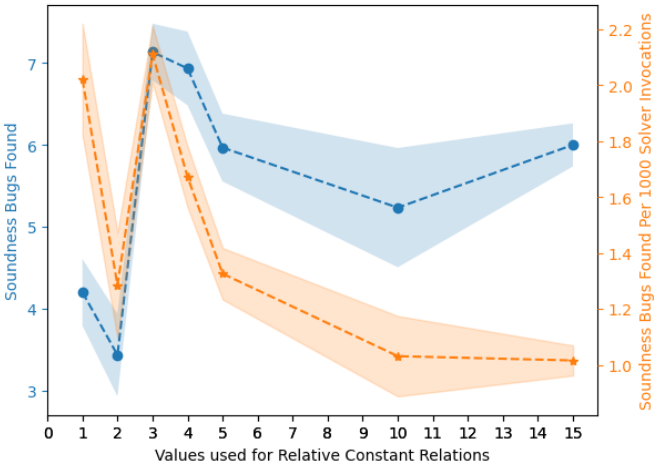
a) The Potential of Value Mutation: We observe that, again, of the known bugs we tested, 21 of them were trivial to reproduce. However, the remaining 17 bugs are **clearly** influenced by the constant values present in their triggering formula, as changing these values hides the buggy behavior. This result further reinforces that value mutation is able to find bugs from non-triggering seed formulas and has potential as a testing methodology.

b) Parameter Optimization: While most of the modifications discussed in Section V can only be either enabled or disabled, both Multi-Constraint Enforcement (Section V-E) and Relations Between Constants (Section V-F) take an integer parameter. For the former, this parameter corresponds to the number of constraints to simultaneously enforce, and, for the latter, it refers to the number of constants to consider for

Fig. 5: Parameter Optimization for ValFuzz Modifications



(a) Soundness Bug Finding Rate and Totals Against the Number of Constraints Enforced (n=30)



(b) Soundness Bug Finding Rate and Totals Against the Number of Constants Used in Relational Constraints (n=30)

such relations. The optimal values for these parameters are likely dependent on both characteristics of the files in the seed corpus and on the other modifications that are enabled in ValFuzz (among other factors). For the evaluation of ValFuzz on the data-set of known bugs, we first tested each of these parameters individually at several levels to obtain an estimate of the optimal values. The results of these tests are presented in Figures 5a and 5b.

From Figure 5a, we see that enforcing roughly seven constraints simultaneously found both the most bugs and was the most efficient for this particular data-set. Looking at Figure 5b, we can see that using just three constants for relational constraints seemed to give the best results as well. As a result, we used these values in the final evaluation.

c) ValFuzz Effectiveness: At a high level, we can see from the results in Table V that our basic approach with skolemized quantifiers (SKOLU) performs slightly, but still statistically significantly, better on average than the random-

TABLE V: Results of Value Mutation Fuzzing on Skeletons of Known Bugs, Mean and Standard Deviation (n=30). Efficiencies are given per 1000 solver invocations as described in Section VI-B

Strategy	Soundness Bugs	Total Bugs	Soundness Efficiency	Total Bug Efficiency
RBASE	3.53 ± 0.57	39.30 ± 1.58	0.61 ± 0.14	4.41 ± 0.38
SKOLU	4.20 ± 0.41	35.00 ± 1.17	2.02 ± 0.20	8.16 ± 0.55
ADOMAIN	6.47 ± 0.57	36.63 ± 1.45	1.89 ± 0.17	5.45 ± 0.40
ADOMAINE	4.50 ± 0.63	23.03 ± 1.03	1.15 ± 0.16	3.84 ± 0.26
CPOG	4.03 ± 0.61	38.93 ± 1.17	1.41 ± 0.21	6.62 ± 0.41
EFFINAL	3.33 ± 0.55	30.97 ± 0.93	1.12 ± 0.26	6.24 ± 0.44
LEAFOPT	3.13 ± 0.35	34.90 ± 0.92	1.95 ± 0.22	10.50 ± 0.58
MULTIEF7	5.07 ± 0.83	36.70 ± 1.49	2.23 ± 0.36	8.22 ± 0.68
NOSKOLE	3.33 ± 0.48	33.30 ± 0.95	1.68 ± 0.24	7.70 ± 0.46
RELC3	7.13 ± 0.35	38.97 ± 0.76	2.11 ± 0.10	5.91 ± 0.23
BLENDA	5.57 ± 0.63	43.47 ± 1.83	1.08 ± 0.12	4.45 ± 0.36
BLENDB	8.87 ± 1.17	42.87 ± 1.59	1.71 ± 0.25	4.72 ± 0.33
BLENDBLEAF	7.03 ± 1.03	42.37 ± 1.56	1.34 ± 0.19	4.53 ± 0.32

ized baseline (RBASE) at finding critical soundness bugs, both in absolute terms and in terms of efficiency (p-values both less than 0.0001). With respect to finding all kinds of bugs, our approach finds fewer in absolute terms on average, but is still significantly more efficient at finding these issues. Augmenting our Boolean Sub-Expression Coercion in ValFuzz with various refinements tended to produce improvements in either efficiency or the absolute number of soundness bugs found. The two notable exceptions being appending constraints to the final formula (EFFINAL) and copying, rather than skolemizing, quantifiers (NOSKOLE), both of which performed worse than our core approach. Interestingly, we also observed that the leaf optimization actually significantly reduces the efficiency of ValFuzz with respect to both soundness and overall bugs when further refinements are also added to our basic approach (BLENDBLEAF), despite that it seems to maintain and improve these efficiencies when combined only with the basic approach (LEAFOPT). Another notable result was that using only variables from the original formula to generate abstract domain constraints (ADOMAIN) performed significantly better than using sub-expressions to generate such constraints (ADOMAINE).

Figures 6a and 6b show a subset of our results presented more visually. Here we focus only on critical soundness issues. From these plots, we see that by adding only some refinements (BLENDA), ValFuzz outperforms completely randomized value mutation in all metrics. Using additional (BLENDB) refinements, we are able to find even more soundness bugs while still maintaining an efficiency far above that of the randomized baseline.

In summary, ValFuzz was very effective at finding non-soundness issues, re-triggering roughly the same number of bugs we are able to reproduce with the original formulas consistently. ValFuzz even found some bugs that were not reproduced with the original formulas from the bug report on our evaluation machine. Soundness issues, however, appear to be much more difficult to reproduce. Anecdotally, this is supported by our experience running ValFuzz on current versions of the solvers as discussed in the previous section; we reproduced many (>20) recent non-soundness issues in parallel as they were reported by others on the Z3 and CVC4 issue trackers, but very few soundness issues (1). We suspect the reasoning behind this lag in performance is due to the extremely large size of the state space being searched during value mutation. OpFuzz, for example, performs a very similar transformation to value mutation, replacing operators with new operators of the same type [31]. The range of possible valid operators for a given position in a formula, however, is relatively small compared to, for example, the number of integers that can fit in an integer constant hole during value mutation. While ValFuzz has clearly made some progress and is searching this space more effectively than a randomized baseline, finding only roughly 7-11 out of 19 known reproducible soundness issues also shows that there is significant room for further innovation.

VII. Future Work

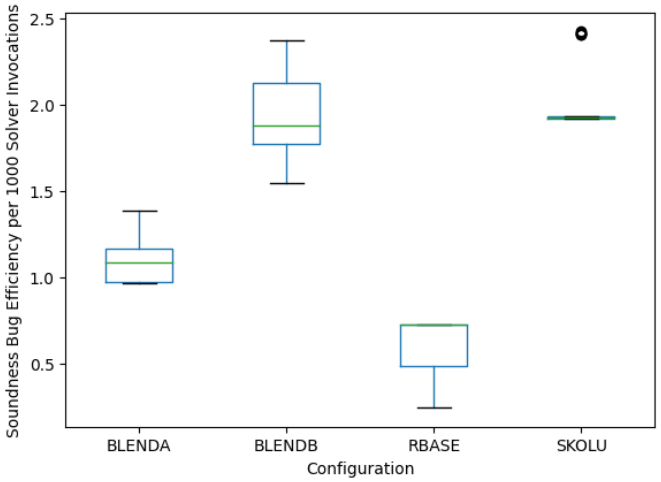
Given that our evaluation indicates further potential for Value Mutation Testing, in the following section, we speculate as to possible directions for future research in this area.

1) Model Sampling: Our implementation of ValFuzz only utilizes a single model for each solver per enforced constraint in Steps 4 and 5 of Boolean Sub-Expression Coercion. Leveraging recent work in the area of model sampling could allow ValFuzz to rapidly test several constant values that satisfy the enforced constraints [13, 14]. These extra test cases could more thoroughly test each enforcement, which may lead to an increase in fault discovery rate.

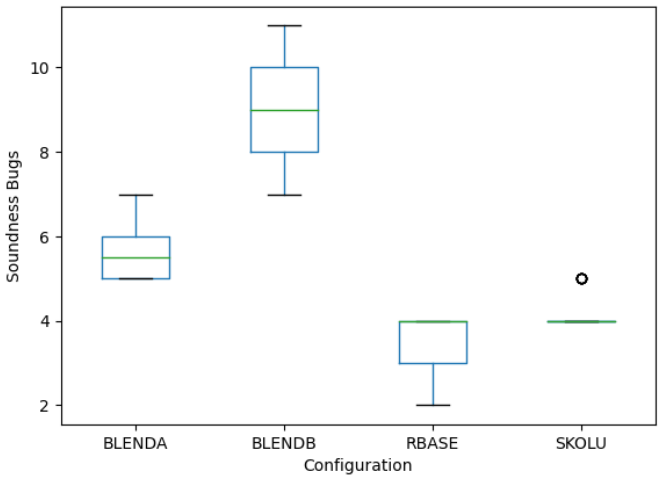
2) MAX-Sat: One feature of modern SMT Solvers is that of *maximum satisfaction*, in which satisfying assignments are directly optimized by the solver relative to a specified objective function [5]. It is possible that utilizing this feature to select value mutations could prove even more effective at finding interesting constant values.

3) Blended Approaches: Value Mutation could be used in conjunction with other successful test case generation approaches such as FusionFuzz [32] as a way to exercise a generated formula skeleton more exhaustively than just a single test case.

Fig. 6: Effectiveness of ValFuzz, Selected Configurations



(a) Soundness Bug Efficiency for Various Value Mutation Strategies (n=30)



(b) Number of Soundness Bugs Found for Various Value Mutation Strategies (n=30)

VIII. Conclusions

We have shown Value Mutation to be an effective method of finding bugs in real-world SMT solver applications. Our implementation of this approach, ValFuzz, found 11 bugs in two state-of-the-art SMT solvers, including one critical soundness bug not detected by previous approaches. Furthermore, using a data-set of previously reported bugs, we have shown that our approach to value mutation is more efficient and more effective at finding bugs than a baseline randomized value mutation approach. Still, while we know that value mutation fuzzers are capable of finding bugs, there is considerable room for improvement and experimentation, especially when it comes to uncovering soundness issues.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Nikolaj Bjørner and Anh-Dung Phan. νz-maximal satisfaction with z3. *Scss*, 30:1–9, 2014.

[6] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In *International Conference on Computer Aided Verification*, pages 45–51. Springer, 2018.

[7] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 2020.

[8] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584843. doi: 10.1145/1670412.1670413. URL `https://doi.org/10.1145/1670412.1670413`.

[9] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *International Conference on Software Engineering (ICSE), 2020*. ETH Zurich, 2020.

[10] Xia Cai and Michael R. Lyu. The effect of code coverage on fault detection under different testing profiles. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083288. URL `https://doi.org/10.1145/1082983.1083288`.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools*

and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.

[13] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of sat solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 549–559. IEEE, 2018.

[14] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. Guidedsampler: Coverage-guided sampling of smt solutions. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 203–211. IEEE, 2019.

[15] Levent Erkök. Smt based verification in haskell, Sep 2020. URL http://leventerkok.github.io/sbv/.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065036. URL https://doi.org/10.1145/1064978.1065036.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10 (1):20–27, 2012.

[18] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[19] Uri Juhasz, Ioannis T Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[20] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. *arXiv preprint arXiv:2004.05934*, 2020.

[21] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[22] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963. ISSN 0001-0782. doi: 10.1145/366246.366248. URL https://doi.org/10.1145/366246.366248.

[23] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697. doi: 10.1145/3274694.3274743. URL https://doi.org/10.1145/3274694.3274743.

[24] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[25] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in smt. In *International Conference on Automated Deduction*, pages 377–391. Springer, 2013.

[26] Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.

[27] Joseph Scott, Federico Mora, and Vijay Ganesh. Banditfuzz: Fuzzing smt solvers with reinforcement learning. 2020.

[28] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[29] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[30] Ole Tange et al. Gnu parallel-the command-line power tool. *The USENIX Magazine*, 36(1):42–47, 2011.

[31] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware mutations for testing smt solvers. *arXiv preprint arXiv:2004.08799*, 2020.

[32] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *PLDI*, pages 718–730, 2020.

[33] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

APPENDIX

A. Glossary of Abbreviations

ADOMAIN – Use abstract domain constraints in the meta-formula for variables as described in Section V-A

ADOMAINE – Use abstract domain constraints in the meta-formula on all sub-expressions as in Section V-A

BLENDA – A combination of ADOMAIN and CPOG

BLENDB – A combination of ADOMAIN, RELC3, and MULTIEF7

BLENDBLEAF – A combination of ADOMAIN, RELC3, MULTIEF7, and LEAFOPT

CPOG – Copy the original formula and its negation into two separate meta-formulas, as in Section V-D

EFFINAL – Enforce constraints from ϕ^{test} in the final ϕ^{sub} formula as described in Section V-G.

LEAFOPT – Use the leaf-boolean-sub-expression-optimization as described in section as in Section V-B

MULTIEF(N) – Enforce N constraints in the meta-formula simultaneously, as described in Section V-E

NOSKOLE – Copy existential quantifiers (in addition to universal quantifiers) as described in Section V-C

RBASE – A randomized Value Mutation Fuzzer used as a baseline to evaluate the effectiveness of Boolean Sub-Expression Coercion and our other modifications to this approach

RELC(N) – Use N constants for relative constraints as in Section V-F

SKOLU – Skolemize universal quantifiers (in addition to existential quantifiers) as described in Section V-C.

B. Solver Flags Used

Randomization Flags Z3:

```
smt.random_seed={}
smt.arith.random_initial_value=true
smt.phase_selection=5
```

Randomization Flags CVC4

```
--random-seed {}
--seed {}
```

Basic Flags Z3

```
model_validate=true
T:{}


smt.arith.solver={}
smt.ematching=false
rewriter.flat=false
smt.string_solver=z3str3
```

Deprecated Flags Z3 (Used to reproduce previously reported bugs as in Section VI-B):

```
rewriter.udiv2mul=true
```

Basic Flags CVC4

```
--produce-models
--strings-exp
--check-models
--incremental
--ho-elim
--check-unsat-cores
--unconstrained-simp
--dump-models
--dump-unsat-cores
--dump-unsat-cores-full
```

CVC4 Validation Flag:

```
--parse-only
```

C. Github Issues for Bugs Discussed in This Paper

Figure 1a: https://github.com/Z3Prover/z3/issues/4153
Figure 4b: https://github.com/Z3Prover/z3/issues/4470
Listing 1: https://github.com/Z3Prover/z3/issues/4607