Towards SMT Solver Stability via Input Normalization

Daneshvar Amrollahi*, Mathias Preiner*, Aina Niemetz*, Andrew Reynolds†,
Moses Charikar*, Cesare Tinelli†, Clark Barrett*
*Stanford University, USA
{daneshvar, preiner, niemetz, moses, barrett}@cs.stanford.edu
†The University of Iowa, USA
{andrew-reynolds, cesare-tinelli}@uiowa.edu

arXiv:2410.22419v2 [cs.LO] 15 May 2025

***Abstract*—In many applications, SMT solvers are utilized to solve similar or identical tasks over time. Significant variations in performance due to small changes in the input are not uncommon and lead to frustration for users. This sort of *stability* problem represents an important usability challenge for SMT solvers. We introduce an approach for mitigating the stability problem based on normalizing solver inputs. We show that a perfect normalizing algorithm exists but is computationally expensive. We then describe an approximate algorithm and evaluate it on a set of benchmarks from related work, as well as a large set of benchmarks sampled from SMT-LIB. Our evaluation shows that our approximate normalizer reduces runtime variability with minimal overhead and is able to normalize a large class of mutated benchmarks to a unique normal form.**

I. Introduction

SMT solvers are used to solve a large variety of problems in academia and industry [4], [14], [21]. As these solvers are integrated into more and more workflows, they are increasingly used in situations where there are many calls to a solver with identical or similar queries. For example, a software verification tool may run a regression suite nightly to check that some software meets its specification, and for the most part, this nightly run does not differ much, if at all, from the previous night's run. A common pain point in such situations is that SMT solver performance can vary significantly, even if there are only minor changes. This has been termed the *stability* problem: queries that are semantically similar or identical may require vastly different amounts of time to solve. Or, even worse, some minor changes may result in a formerly solved query not being solved at all.

As part of an NSF-supported project [3], we spoke with a large number of SMT stakeholders and cataloged and ranked over 100 different issues, based on how often they came up in interviews. In this ranking, instability was ranked as the second-highest area of concern (behind only requests for better diagnostic output when the solver is unable to solve a problem). The instability concern was also highlighted in a keynote talk by Neha Rungta at CAV 2022 [21] on the use of SMT solvers at Amazon Web Services and has been identified by another senior manager at Amazon as one of two top priorities for improving their SMT solving workflows [13].

There are two kinds of changes that can introduce instability: changes to the input and changes to the solver. While both are important, this paper focuses on the first: *changes to the input*. In other words, we are interested in reducing the sensitivity of solving time to minor changes in an input formula, especially if those changes are semantics-preserving.

Two common misconceptions about stability are worth noting. First, instability should not be confused with poor solver performance. Improving solver performance is an important and worthy goal; however, it is orthogonal to addressing the stability problem. Our focus is on making solver performance more *consistent* in the face of mutations to its input. Importantly, many users in our study reported that they would welcome *any* improvement to stability, even if it came at the cost of a degredation in performance. The other misconception is the failure to recognize that the stability problem is linked to computational complexity. Because the SMT problem is NP-hard (or worse, depending on the theory being used), solvers try to guide worst-case exponential algorithms in such a way that solutions are found quickly when possible. However, even a small change in the input can trigger a different search path, which could result in an exponentially worse (or better) runtime. This is similar to the well-known "butterfly effect" in chaos theory. Despite this, as we show in this paper, there *are* steps that can be taken to *reduce* instability.

SMT solvers typically already use deterministic data structures and algorithms to eliminate easily avoidable sources of instability. In a sequential execution setting, the remaining source of variability is their heuristics, primarily their branching heuristics, which typically break ties based on the *order* in which declarations and assertions appear in the input formula. Our key idea is to reduce this variability by using the *structure of the input problem* to determine the order of declarations and assertions. Our approach uses the principle of *normalization*: we attempt to map classes of semantically equivalent inputs to the same normal form. Note that if perfect normalization could be achieved in this setting, it would result in perfect stability, with respect to the semantically equivalent input classes, as the input to the core solver would be the same in every case. We present an approach aimed at getting closer to that ideal without introducing too much overhead. A notable feature of our approach is that it is agnostic to the underlying solver. It can thus be used to improve the stability of any SMT solver with respect to changes in the input.

We consider normalization with respect to a set of basic, semantics-preserving transformations: (i) reordering of assertions; (ii) reordering of operands of commutative operators; (iii) reordering and renaming of user-defined symbol declarations; and (iv) replacing anti-symmetric operators by their converse. These transformations are representative of the kinds of changes that happen in the real-world scenarios motivating our work. For example, in a program verification workflow, if a function is moved or renamed, this could result in a reordering of declarations and assertions, a different name for a user-defined symbol, or both. These transformations are also a superset of the solver input transformations considered in previous work on *measuring* instability [26]. We will refer to these transformations as *mutations* and to inputs that have been transformed this way as *mutated* inputs. We address the following research questions:

1) Is it possible to design a normalizing algorithm that utilizes these mutations to map all mutated variants to a single unique normal form?
2) If such an algorithm exists, what is its time complexity?
3) How closely can an efficient algorithm approximate the ideal algorithm?

After covering some background in Section II, we formalize the problem in Section III and answer the first two questions, showing that such an algorithm does indeed exist but is as hard as graph isomorphism. We provide an answer to the third question in the remainder of the paper. Section IV introduces an algorithm that approximates the ideal algorithm, and Section V presents an evaluation of our implementation, showing that it significantly improves SMT solver stability on benchmarks from [26] and from the SMT-LIB benchmark library [20]. Finally, Section VI concludes.

Related work. The importance of the issue of stability in SMT solving has been raised in other work [8], [10], [12], [17]. Dodds [8], highlights the problem of proof fragility under changes in verification tools. Ferraiuolo et al. [10] mention proof instability as *the most frustrating recurring problem*, especially when proof complexity increases as a result of reasoning about procedures with many instructions and complex specifications. In Hawblitzel et al. [12], verification instability is observed in large formulas and non-linear arithmetic due to different options for applicable heuristics. Leino et al. [17] identify *matching loops*—caused by certain forms of quantifiers that lead an SMT solver to repeatedly instantiate a limited set of quantified formulas—as a key factor contributing to instability in verification times, and describe techniques to detect and prevent them.

More relevant to our work is the work of Zhou et al. [26], [25]. In [26], they pioneer an effort to detect and quantify instability and introduce a tool for this task called Mariposa. They show that mainstream SMT solvers such as Z3 [7] and cvc5 [1] exhibit instability on a set of F* [22] and Dafny [16] benchmarks [5], [10], [11], [18], [19], [23]. They consider the benchmark-modifying mutations of symbol renaming and assertion reordering, as well as solver-modifying mutations via the use of different random seeds, and use a statistical approach to identify instability arising from these mutations. We include an evaluation of our technique on their benchmarks, with the notable difference that we do not include solver-modifying mutations. This is because we aim to improve stability with respect to input changes, rather than across changes to a solver. In [25], Zhou et al. identify irrelevant context in a query as one source of instability and propose a novel approach to filter out such context to improve solver stability. This approach is complementary to our own, and combining the two is an interesting direction for future work.

Also closely related is the work of Weber [24], which introduces a normalizer designed to reverse the effects of mutations used to *scramble* benchmarks for the SMT-COMP competition at the time. However, the normalizer introduced in [24] was specifically designed to expose a weakness in the scrambling algorithm, namely that symbols were renamed but not reordered. This weakness makes it possible to achieve perfect normalization efficiently, which their algorithm does. However, in the presence of the more expressive and realistic mutations we consider, their normalizer performs poorly, as we show in Section V.

II. Background

We work in the context of many-sorted logic (e.g., [9]), where we assume an infinite set of variables of each sort and the usual notions of signatures, terms, formulas, assignments, and interpretations. We assume a signature Σ consisting of sort symbols and sorted function symbols. It is convenient to assume that Σ has a distinguished sort `Bool`, for the Booleans, and to represent relation symbols as function symbols whose return sort is `Bool`. We also assume the signature includes equality. Symbols in Σ are partitioned into *theory symbols* (e.g., $=, \wedge, \vee, +, -, 0, 1$) and *user-defined symbols* (e.g., f, g, x, y). We assume some *background theory* restricts the theory symbols to have fixed interpretations, whereas the interpretation of user-defined symbols is left unrestricted.

We represent formulas as finite *sequences* of symbols in prefix notation, where each symbol is either a theory symbol or a user-defined symbol. This causes no ambiguity when each symbol has a fixed arity. If S is a sequence $\langle s_1, \ldots, s_n \rangle$, we write $|S|$ to denote n, the length of the sequence, and S_i to denote the i^{th} element of the sequence. We write $s \in S$ to mean that s occurs in the sequence S, and write $S \circ S'$ for the sequence obtained by appending S' at the end of S. We write $user(S)$ to mean the sequence obtained by deleting all theory symbols in S, e.g., $user(\langle x, +, y, -, x \rangle) = \langle x, y, x \rangle$. We denote the set of integers between m and n inclusive, where $n \geq m$, as $[m, n]$. And we abbreviate $[1, n]$ as $[n]$.

A formula in the SMT-LIB 2.6 format [2] is shown in Figure 1. We use this as a running example throughout the paper. The example includes arithmetic theory symbols, a user-defined function f, and user-defined constants v, w, x, y, and z. We will refer to the sequence of six assertions in the running example as $\langle \beta_1, \ldots \beta_6 \rangle$. Figure 2 shows two different representations of the first two assertions in the example: the

```
(set-logic QF_UFLIA)

(declare-fun f (Int) Int)  (declare-const v Int)
(declare-const w Int)      (declare-const x Int)
(declare-const y Int)      (declare-const z Int)

(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))
(assert (>= (+ (f y) x) (- w 12)))
(assert (< (+ y x) (* x x)))
(assert (< (+ x y) (* y y)))
(assert (< (+ y x) (* y v)))

(check-sat)
```

Fig. 1: Running Example.

```
(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))
```

```
(assert (> (* u4 u2) (+ u2 u3)))
(assert (>= (+ (g u2) u3) (- u1 12)))
```

Fig. 2: Two different mutated versions of the same assertions.

first is from the example and the second is a mutation of it. In particular, the order of the assertions has been swapped, the operands of the $*$ operator have been reordered, the user-defined symbols have been renamed (with `w`, `x`, `y`, and `z` renamed to `u1` through `u4`, respectively, and `f` renamed to `g`), and the assertion using $<$ has been rewriiten to use $>$.

III. Formalization

Consider again the first two assertions from the example. Here, the user-defined symbols are $\{f,\ x,\ y,\ v,\ z\}$, and the theory symbols are $\{>=,\ <,\ +,\ -,\ *,\ 12\}$. When written as a sequence in prefix notation, β_1 is $\langle >=, +, f, x, y, -, v, 12\rangle$. Similarly, β_2 is $\langle <, +, x, y, *, x, z\rangle$.

Recall that mutations include four operations: (i) reordering assertions; (ii) reordering operands of commutative operators; (iii) reordering and renaming symbols; and (iv) replacing anti-symmetric operators. We defer (iv) to the end of this section, as it can easily be handled separately. To formalize the others, we introduce some definitions.

Definition III.1 (Shuffle Set)**.** The *shuffle set* of a sequence A is defined as $S(A) = \{A' \mid A' \text{ is a permutation of } A\}$.

For example, $S(\langle\beta_1, \beta_2\rangle) = \{\langle\beta_1, \beta_2\rangle, \langle\beta_2, \beta_1\rangle\}$.

Definition III.2 (Commutative Reordering Set)**.** Let α be a formula, possibly containing (binary) commutative operators. The *commutative reordering set* of α is defined as $C(\alpha) = \{\alpha' \mid \alpha' \text{ is the result of swapping the operands of zero or more of these commutative operators in } \alpha\}$. For a sequence A of n formulas, $C(A)$ is the set of all sequences $\langle\alpha'_1, \ldots, \alpha'_n\rangle$ where for each $i \in [n]$, $\alpha'_i \in C(\alpha_i)$. If X is a set (containing either formulas or sequences of formulas), then $C(X) = \{x' \mid x' \in C(x) \text{ for some } x \in X\}$.

For example, $C(\beta_1) = \{\beta_1, \langle >=, +, y, f, x, -, v, 12\rangle\}$. Note that there is only one entry in $C(\beta_1)$ besides β_1 itself, because $+$ is the only commutative operator appearing in β_1. On the other hand, there are four elements in $C(\beta_2)$, since β_2 has two commutative operators, $+$ and $*$. Consequently, $C(\langle\beta_1, \beta_2\rangle)$ has eight elements, representing all combinations of one formula each from $C(\beta_1)$ and $C(\beta_2)$.

Definition III.3 (Pattern)**.** Let α be a formula. The *pattern* of α, written $P(\alpha)$, is a sequence of the same length as α, defined for each $i \in [|\alpha|]$ by

$$P(\alpha)_i = \begin{cases} \alpha_i & \text{if } \alpha_i \text{ is a theory symbol,} \\ @1 & \text{if } \alpha_i \text{ is the first user-defined symbol appearing in } \alpha, \\ P(\alpha)_j & \text{if } \alpha_i \text{ is a user-defined symbol and there exists } j \in [i-1] \text{ with } \alpha_j = \alpha_i, \\ @k & \text{otherwise, where } k = 1 + \big|\{\alpha_j \mid j \in [i-1],\ \alpha_j \text{ user-defined}\}\big|. \end{cases}$$

For convenience, we assume that each symbol $@k$ is a fresh constant[1] of the same sort as the symbol it is replacing, so that if α is well-sorted, then so is $P(\alpha)$. We call introduced symbols starting with "@" *pattern symbols* and sequences in the co-domain of P *patterns*. We define a total order $\prec$ on patterns to be the lexicographic order induced by some total order on formula symbols.[2] For our example assertions, we have $P(\beta_1) = \langle >=, +, @1, @2, @3, -, @4, 12\rangle$ and $P(\beta_2) = \langle <, +, @1, @2, *, @1, @3\rangle$.

We lift this notation to sequences and sets of sequences. To explain how, we need two more definitions.

Definition III.4. Given a sequence of formulas $A = \langle\alpha_1, \ldots, \alpha_n\rangle$, the *conjoining* of A, $Conj(A)$ is the formula $\langle\wedge\rangle \circ \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_n$. Similarly, given a formula $\langle\wedge\rangle \circ \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_n$, where α_i is a formula for $i \in [n]$, the *unconjoining* of α, written $Unconj(\alpha)$ is the formula sequence $\langle\alpha_1, \ldots, \alpha_n\rangle$.

For a sequence A of formulas, we define $P(A) = Unconj(P(Conj(A)))$. If X is a set (containing either formulas or sequences of formulas), then $P(X) = \{P(x) \mid x \in X\}$. We similarly lift $\prec$ to sequences of formulas: if A and A' are sequences of patterns, then $A \prec A'$ iff $\alpha_1 \circ \cdots \circ \alpha_{|A|} \prec \alpha'_1 \circ \cdots \circ \alpha'_{|A'|}$. We next define renaming and normalization.

Definition III.5 (Renaming)**.** Let V be a set of variables. A *renaming* R is an injective function from pattern symbols to V. For a formula α, $R(\alpha)$ is defined to be a sequence of the same size as α, defined as follows:

$$R(\alpha)_i = \begin{cases} \alpha_i & \text{if } \alpha_i \text{ is a theory symbol,} \\ R(\alpha_i) & \text{if } \alpha_i \text{ is a pattern symbol.} \end{cases}$$

For a sequence of formulas $A = \langle\alpha_1, \ldots, \alpha_n\rangle$, $R(A) = \langle R(\alpha_1), \ldots, R(\alpha_n)\rangle$. If X is a set (containing either formulas or sequences of formulas), then $R(X) = \{R(x) \mid x \in X\}$.[3]

[1]The SMT-LIB 2.6 standard reserves symbols starting with @ for internal use by solvers, so this assumption is a reasonable one.

[2]An obvious choice for this order (and the one we use) is the lexicographic order on the string representations of theory and pattern symbols.

[3]When representing an assertion sequence as an SMT-LIB 2.6 script, we declare user-defined symbols in lexicographic order, so a renaming can affect the order of declarations.

Definition III.6 (Normalizing Function)**.** A function N from sequences of formulas to sequences of formulas is said to be *normalizing* if, for every sequence A of formulas:

1) $N(A) = R(A')$ for some $A' \in P(C(S(A)))$ and some renaming R; and
2) if $M_1 = R_1(M_1')$ and $M_2 = R_2(M_2')$, with R_1, R_2 renamings and with $M_1', M_2' \in P(C(S(A)))$, then $N(M_1) = N(M_2)$.

We can now introduce our first research question: does there exist a normalizing function? We show that the question can be answered affirmatively.

Definition III.7 (Normalizing Function)**.** Let $\mathcal{N}$ be defined as follows. Given a sequence of formulas A, let $\mathcal{N}(A)$ be the $\prec$-minimal element of $P(C(S(A)))$.

Theorem III.8. *Function $\mathcal{N}$ is normalizing.*

Due to lack of space, proofs for this and other theorems in this paper are omitted.[4]

A. Complexity

We have shown the existence of a normalizing function. The next question is whether normalization can be done efficiently. An informal argument that computing the normalization of an arbitrary set of formulas is at least as hard as graph isomorphism can be found in [15], and a formal proof can be found in [24]. The question of whether graph isomorphism can be solved in polynomial time is a long-standing open problem.

Theorem III.9. *Let N be a normalizing function. Then, computing $N(A)$ for an arbitrary A is as hard as solving graph isomorphism.*

The use of C is not essential for the proof to succeed. Normalizing just the result of shuffling and renaming is already as hard as graph isomorphism.

B. Anti-symmetric Operators

As mentioned above, mutations can randomly replace anti-symmetric operators with their dual operator. For example, `(< (+ x y) (* x z))` could be changed to `(> (* x z) (+ x y) )`. In general, we allow mutations that transform expressions of the form A *op* B into B *op'* A, where (*op*, *op'*) pairs include: (>, <), (>=, <=), (`bvugt`, `bvult`), (`bvuge`, `bvule`), ...

A normalization algorithm can easily handle anti-symmetric operators, simply by choosing one representative operator for each pair and forcing all assertions to use only the chosen operators. For example, if the first operator in each pair listed above is the chosen one, then Figure 3 shows the result of normalizing the first two assertions in our running example. Notice that the second assertion is modified by replacing < with > and swapping the operands but the first assertion is unchanged because it is already using the chosen operator.

[4]For the reviewers' convenience, the proofs are provided in an appendix.

```
(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))
```

```
(assert (>= (+ (f x) y) (- v 12)))
(assert (> (* x z) (+ x y)))
```

Fig. 3: Original (top) and normalized (bottom) assertions.

IV. Approximating a Normalization Algorithm

As a first step towards a general practical algorithm for normalization, we describe a heuristic procedure designed to handle two of the four mutations: shuffling and renaming. We leave the handling of the other operations to future work. We expect support for normalizing antisymmetric operator replacement to be straightforward (as described above), while normalizing commutative operand swapping will be more challenging. Note that shuffling and renaming are also the two operations used to mutate benchmarks in the Mariposa work (the closest related work) [26]. We describe our algorithm at a conceptual level here, and discuss several optimizations necessary to make it work well in practice in Section V-C. Our algorithm consists of three steps: (i) Sorting the assertions; (ii) Renaming all symbols; and (iii) Sorting the assertions again. In the rest of this section, we discuss these steps in detail.

Sorting the assertions. Step one is to sort the assertions. The challenge is to do this in a way that does not depend on the names of user-defined symbols. The key idea is to use patterns. In particular, to order assertions α and α', we can compare $P(\alpha)$ and $P(\alpha')$ using the $\prec$ order. For instance, considering agian assertions from our running example, β_5 is $\langle <, +, x, y, *, y, y\rangle$, and β_6 is $\langle <, +, y, x, *, y, v\rangle$, so we have $P(\beta_5) = \langle <, +, @1, @2, *, @2, @2\rangle$ and $P(\beta_6) = \langle <, +, @1, @2, *, @1, 3\rangle$. The first difference in the patterns is in the sixth position. Assuming @1 is ordered before @2, we have that $P(\beta_6) \prec P(\beta_5)$, so we can conclude that β_6 should be placed before β_5.

Note, however, that it is possible for two formulas to have the same pattern. Thus, after sorting according to patterns, we obtain an ordered list of equivalence classes $EC_1, \cdots, EC_n$ with the following features:

1) α and α' belong to the same equivalence class iff $P(\alpha) = P(\alpha')$.
2) $\alpha \in EC_i$ and $\alpha' \in EC_j$ where $i < j$ iff $P(\alpha) < P(\alpha')$.

The next question is whether we can easily order the assertions belonging to the same equivalence class. We give an efficient approximation method. For this, we need the notions of role and super-pattern.

Definition IV.1 (Role)**.** The role of a symbol s in a formula α, denoted $role(s, \alpha)$, is 0 if $s \notin \alpha$ and the index of the earliest occurrence of s in $user(\alpha)$, otherwise. The role of s in a set of formulas is the multiset consisting of all the roles played by s in the formulas in the set.

For example, consider the role of y in β_5. First of all, we compute $user(\beta_5)$, which is $\langle x, y, y, y\rangle$. We can then see that y occurs first at the second position, so $role(y, \beta_5) = 2$. Similarly, $role(x, \{\beta_4, \beta_5, \beta_6\}) = \{1, 1, 2\}$.

```
(assert (< (+ x y) (* x z)))
(assert (< (+ y x) (* y v)))

(assert (< (+ y x) (* x x)))
(assert (< (+ x y) (* y y)))

(assert (>= (+ (f x) y) (- v 12)))
(assert (>= (+ (f y) x) (- w 12)))
```

Fig. 4: Assertions sorted by pattern.

```
(assert (< (+ X1 X2 ) (* X1 X3 )))
(assert (< (+ X2 X1 ) (* X2 X4 )))

(assert (< (+ X2 X1 ) (* X1 X1 )))
(assert (< (+ X1 X2 ) (* X2 X2 )))

(assert (>= (+ (X5 X1 ) X2 ) (- X4 12)))
(assert (>= (+ (X5 X2 ) X1 ) (- X6 12)))
```

Fig. 5: Assertions after sorting and renaming.

Definition IV.2 (Super-pattern)**.** The super-pattern of a symbol s over a sequence X of sets $X_1, \ldots, X_n$, denoted $SP(s, X)$, is the sequence of roles of the symbol in each set: $SP(s, X) = \langle role(s, X_1), role(s, X_2), \ldots, role(s, X_n)\rangle$.

Let EC be a sequence of formula equivalence classes. The super-pattern of EC captures the role of a symbol across all equivalence classes, while treating the formulas in each equivalence class as unordered.

To illustrate, recall the example from Figure 1. Figure 4 shows the result of sorting the assertions by pattern, resulting in three equivalence classes, each separated by an empty line. The patterns of the equivalence classes in $EC = \{EC_1, \ldots, EC_3\}$, from top to bottom, are as follows:

EC_1: $\langle <, +, @1, @2, *, @1, @3\rangle$
EC_2: $\langle <, +, @1, @2, *, @2, @2\rangle$
EC_3: $\langle >=, +, @1, @2, @3, -, @4, 12\rangle$

Now, suppose we want to order the formulas in EC_3. We compare the super-patterns of the first different pair of user-defined symbols, in this case, x and y. The roles of x throughout the equivalence classes are:

$role(x, EC_1) = \{role(x, \beta_2), role(x, \beta_6)\} = \{1, 2\}$
$role(x, EC_2) = \{role(x, \beta_4), role(x, \beta_5)\} = \{1, 2\}$
$role(x, EC_3) = \{role(x, \beta_1), role(x, \beta_3)\} = \{2, 3\}$

Therefore, applying the definition of super-pattern for x yields $SP(x, EC) = \langle\{1, 2\}, \{1, 2\}, \{2, 3\}\rangle$. It is not hard to see that the super-pattern for y is the same, so the two assertions cannot be distinguished by looking at x and y. The next pair of different user-defined variables also consists of x and y. However, the last pair is v and w. Following the same process, we find that $SP(v, EC) = \langle\{0, 4\}, \{0, 0\}, \{0, 4\}\rangle$ and $SP(w, EC) = \langle\{0, 0\}, \{0, 0\}, \{0, 4\}\rangle$. Now, all we need is a way to order different super-patterns.

Definition IV.3 (Integer multiset order)**.** Given multi-sets of integers m_1 and m_2, $m_1 < m_2$ iff the sequence of nondecreasing elements of m_1 is lexicographically less than the sequence of nondecreasing elements of m_2.

Definition IV.4 (Super-pattern order)**.** For super-patterns s_1 and s_2, $s_1 < s_2$ iff s_1 comes before s_2 when compared using the lexicographic order induced by the integer multiset order.

Thus, when comparing super-patterns, we compare the entries in the sequences one by one using the integer multiset order. The first two entries in $SP(v, EC)$ and $SP(w, EC)$ are $\{0, 4\}$ for v and $\{0, 0\}$ for w. Because $\langle 0, 0\rangle < \langle 0, 4\rangle$, we can conclude that $SP(w, EC) < SP(v, EC)$. Thus, we should switch the order of assertions in the last equivalence class. Similarly, by computing super-patterns for z and v, we can see that we should keep the order of the assertions in the first equivalence class.

It is possible for all of the super-patterns of corresponding symbols in two assertions in the same equivalence class to be the same. In this case, our heuristic algorithm fails, and the assertion order is left unchanged. For example, for EC_2, the only user-defined symbols available for comparison are x and y, and they have the same super-pattern. Thus, we leave these assertions in their original order for now. Algorithm 1 shows the full algorithm for ordering two assertions.

Algorithm 1 Algorithm for ordering assertions A and B given EC, a sequence of equivalence classes (with respect to pattern equality) of assertions.

```
if P(A) ≠ P(B) then
  return P(A) ≺ P(B)
end if
for i ← 1 to |user(A)| do
  u ← user(A)_i
  v ← user(B)_i
  if u = v then
    continue
  end if
  if SP(u, EC) = SP(v, EC) then
    return SP(u, EC) < SP(v, EC)
  end if
end for
return inconclusive
```

Renaming all symbols. After sorting the assertions according to Algorithm 1, we rename all the symbols in the assertions. We use a renaming $\mathbb{R}$ that maps a variable whose pattern symbol is $@k$ to X_k. More precisely, if A is the sequence of assertions after sorting, we replace A with $\mathbb{R}(P(A))$. Figure 5 shows the assertions from our running example after sorting and renaming.

Sorting the assertions again. After renaming, there is one more step that can improve the normalizer. It is based on the observation that within an equivalence class, different assertion orders are possible, depending on the initial order, when all symbols in a pair of assertions have the same super-patterns. This can be partially addressed by lexicographically sorting each equivalence class after renaming. This ensures that if we have two benchmarks for which the first two steps produce the same set of assertions, but in different orders, then these two benchmarks will be normalized the same way.

Looking again at Figure 5, we see that assertions in the first and last equivalence classes are already in sorted order. However, the assertions in equivalence class 2 should be reordered. Recall that in the previous step, we did not have a way to order these assertions, but now there is an unambiguous order for them. It is important to note that our algorithm does not guarantee the normalization property. The reason for this incompleteness is that when assertions cannot be distinguished by super-patterns, there can be different normal forms for benchmarks, even if one is a mutation of the other (via shuffling and renaming). However, our algorithm works well in practice, as we show next.

V. Experiments

We implemented our normalization algorithm in C++. We evaluate its effectiveness on three dimensions: *normalization effectiveness*, *stability*, and *runtime*.

We use two sets of benchmarks. The first set, *smtlib*, is obtained by randomly selecting 50 benchmarks from each family (or all of them, if there are fewer than 50) in the SMT-LIB benchmark library [20]. A family, in this context, consists of all benchmarks in a single leaf directory of the filesystem heirarchy. Note that even though our normalization algorithm as described in Section IV is, in theory, applicable to all of SMT-LIB without limitation, our current implementation does not support the normalization of algebraic datatypes due to implementation-level complexities. We thus excluded benchmarks with algebraic datatypes from our evaluation. Still, *smtlib* contains 41,166 benchmarks from 1,581 benchmark families. We group benchmarks from this set into the (so-called) divisions used by the annual SMT-COMP [6] to keep the number of categories manageable.

The second set, *mariposa*, consists of the benchmarks used in [26] (as provided in [27]), originating from program verification projects written in Dafny [16], Serval [19], and F* [22]. *mariposa* contains 16,622 benchmarks and is divided into six families: Dice$^*_F$ [23] (1536), Komodo$_D$ and Komodo$_S$ [10] (2,054 and 773), VeriBetrKV$_D$ [11] (5,170), VeriBetrKV$_L$ [18] (5,334) and vWasm$_F$ [5] (1,755).

For each benchmark, we produce 10 mutations by randomly applying assertion shuffling and renaming. Our renaming uses a fixed enumeration of names (i.e., a benchmark wth n user-defined symbols always uses the first n of these names), but shuffles the order in which they appear in assertions (and thus the order in which they are declared, since we declare symbols in the order in which they appear). We ran all experiments on a cluster of 48 machines with AMD Ryzen 9 7950X CPUs using a time limit of 60 seconds and a memory limit of 8GB. We used this same time and memory limit to separately limit the mutation step, the normalization step, and the solving step.[5] We do not report results for benchmarks that timed out during mutation or normalization. For the *smtlib* set, we exclude 6 benchmarks that timed out during the mutation step and 85 that timed out during the normalization phase (of these, 58 timed out during parsing, before ever getting to the normalization algorithm), for any of the 10 mutated versions. In *mariposa*, we only exclude 2 benchmarks, both of which timed out during normalization. Overall, this resulted in 41,075 eligible benchmarks in *smtlib* and 16,620 in *mariposa*.

[5]This is long enough to be able to catch cases when the normalizer is slow and short enough to help keep a large evaluation computationally tractable. Using the same timeout for all stages also keeps things simple.

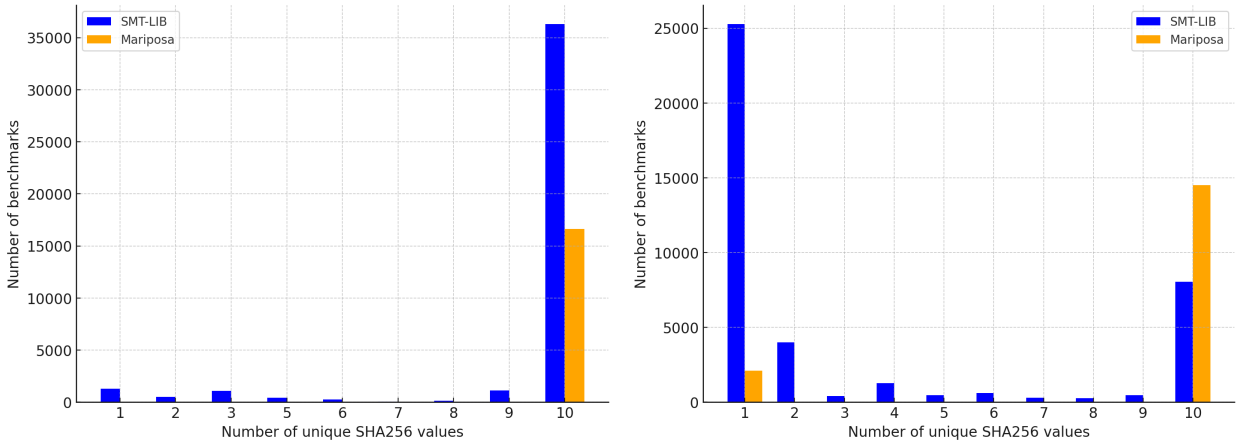
A. Normalization Effectiveness

In our first experiment, we evaluate how closely our implementation of an efficient normalization algorithm approximates the ideal algorithm on both benchmark sets. For this, we measure, for each benchmark: (i) the number of distinct outputs produced by our normalizer for the 10 mutations (the best possible is 1; the worst possible is 10); and (ii) the *similarity* of the 10 normalized outputs, computed as the average percentage of identical lines among all pairs of outputs. Table I shows our results. Each row starts with a category name (the name of the division or family) and a pair of numbers (x/y) denoting the number of excluded benchmarks x vs. the total number of benchmarks y in that category. We then list the number of considered benchmarks (*#Bench*, equal to $y - x$) and the average number of distinct benchmarks produced by the 10 mutations, before normalization (*pre*) and after normalization, without (*wo-SP*) and with (*w-SP*) the use of super-patterns. We next show the average and maximum runtime for the normalization algorithm. Finally, we show the average similarity among the mutated benchmarks, again comparing *pre*, *wo-SP*, and *w-SP*. Figure 6 shows histograms of the number of distinct benchmarks among the 10 mutations before and after (with super-patterns) normalization.

First, we observe that before normalization, we nearly always have 10 distinct versions of a benchmark. Exceptions (e.g., in the QF_Strings category) occur when there are not a sufficient number of user-defined symbols and assertions to create 10 distinct versions. For the *smtlib* benchmark set, normalization always significantly reduces the number of distinct versions of a benchmark, and results with super-patterns always improve over those without super-patterns—sometimes significantly (e.g., in the Equality division).

For other benchmarks, however, especially the QF_Equality+NonLinearArith division and several families in the *mariposa* benchmark set, our normalizer struggles to produce a small number of distinct outputs. Investigating these benchmarks reveals large numbers of assertions with lots of symmetry. In these cases, the super-pattern comparison fails to distinguish between assertions with identical patterns, resulting in different normal forms because of assertions appearing in a non-deterministic order. It is worth highlighting that for these benchmarks, our normalization algorithm still achieves a high level of similarity. As we discuss below, there is also an improvement in stability for these benchmarks. This suggests that full normalization is not necessary to improve stability. Improved similarity is already helpful.

As mentioned above, over all of the benchmarks, only 87 are excluded due to timeouts during normalization, and these are typically very large benchmarks where parsing alone takes



(a) Before Normalization

(b) After Normalization

Fig. 6: Number of distinct benchmarks before and after normalization.

Division (***smtlib***)	(Excluded/Total)	**#Bench**	**#Unique**			**Time (s)**		**Similarity (%)**		
			pre	*wo-SP*	*w-SP*	*Avg*	*Max*	*pre*	*wo-SP*	*w-SP*
Arith	(0/883)	883	9.33	1.05	1.02	0.0010	0.1720	7.07	99.39	99.62
BitVec	(5/793)	788	9.36	1.07	1.03	0.0139	3.5230	5.81	99.01	99.48
Equality	(0/1,570)	1,570	10.00	8.48	2.79	0.0075	0.1020	0.05	47.59	98.48
Equality+LinearArith	(0/2,595)	2,595	9.99	7.34	5.35	0.0030	0.0860	1.29	65.56	86.78
Equality+MachineArith	(2/905)	903	10.00	1.25	1.22	0.0295	3.6140	0.76	97.97	98.18
Equality+NonLinearArith	(0/1,452)	1,452	9.70	2.98	1.64	0.0080	0.4980	3.14	88.89	98.48
FPArith	(0/319)	319	9.48	1.03	1.03	0.0000	0.0010	5.25	99.61	99.61
QF_Bitvec	(42/3,320)	3,278	9.55	4.94	4.78	0.7248	38.7570	4.63	74.17	79.88
QF_Equality	(0/667)	667	10.00	2.63	2.16	0.0306	1.0710	0.73	85.67	93.24
QF_Equality+Bitvec	(20/1,208)	1,188	9.87	4.49	4.44	0.1222	13.3070	2.94	71.07	74.06
QF_Equality+LinearArith	(12/812)	800	9.95	3.66	3.38	0.0590	6.6460	1.58	83.17	89.38
QF_Equality+NonLinearArith	(0/589)	589	10.00	6.73	6.22	0.2908	29.6690	1.76	71.40	85.08
QF_FPArith	(0/4,813)	4,813	9.08	2.80	2.52	0.0002	0.0290	13.17	90.93	92.30
QF_LinearIntArith	(6/10,801)	10,795	9.98	6.35	4.21	0.2728	37.2840	8.62	78.79	88.67
QF_LinearRealArith	(3/1,466)	1,463	9.95	3.72	3.26	0.2230	30.3610	6.77	91.19	93.86
QF_NonLinearIntArith	(0/653)	653	9.88	3.04	2.64	0.0752	4.5400	4.77	94.00	95.69
QF_NonLinearRealArith	(1/3,343)	3,342	9.83	1.83	1.67	0.0620	10.5110	8.42	96.05	97.76
QF_Strings	(0/4,977)	4,977	6.21	2.69	2.45	0.0004	0.0190	44.50	91.75	94.93
Family (***mariposa***)										
Dice$^*_F$	(0/1536)	1536	10.00	10.00	10.00	5.4847	12.4670	0.93	99.45	98.75
Komodo$_D$	(0/2054)	2054	10.00	10.00	10.00	1.2750	3.7500	0.18	34.87	88.28
Komodo$_S$	(2/773)	771	10.00	1.00	1.00	0.0004	0.0790	1.09	100.00	100.00
VeriBetrKV$_D$	(0/5170)	5170	10.00	10.00	10.00	0.0790	9.1740	0.19	18.15	87.30
VeriBetrKV$_L$	(0/5334)	5334	10.00	10.00	10.00	1.2533	9.6420	0.17	12.73	81.85
vWasm$_F$	(0/1755)	1755	10.00	3.16	3.15	0.0657	4.2050	0.01	99.53	99.52

TABLE I: Number of unique outputs, normalization time, and similarity of normalized outputs.

most or all of the time. For the vast majority of the remaining benchmarks, the normalization overhead is extremely low (a fraction of a second on average as shown in Table I). The aggregate overhead, computed as the sum of the normalizing time for all benchmarks divided by sum of the normalization plus solving time for all benchmarks, is less than 0.8%.

We also conducted exploratory experiments on a random subset of *smtlib* (4,461 benchmarks) to evaluate the performance of the normalizer in [24] in terms of uniqueness of the normalized output. We observed that it was able to produce unique outputs for only 12%, of the benchmarks, which is vastly outperformed by our approach (87% on this set). This result, however, is not unexpected since the algorithm in [24] was designed for the purpose of exploiting a specific weakness in the SMT-COMP's scrambling algorithm in use at the time rather than with general-purpose normalization in mind.

B. Stability

In our second experiment, we evaluate how our normalization algorithm affects solver stability. We use two SMT solvers: cvc5 [1] and Z3 [7]. These are natural choices, as both are used extensively and support a wide range of theories. We limit our evaluation to these two solvers, as they are the only solvers that support all of our benchmarks. For the *mariposa* benchmark set, we limit our evaluation to Z3. This is because these benchmarks come from a specific use case targeting Z3 (as observed already by Zhou et al. [26]), and most of them are unsolvable by cvc5.

Division (*smtlib*)	**cvc5 PR-2**				**Z3 PR-2**			
	no norm.		*norm.*		*no norm.*		*norm*	
	avg.	*MAD*	*avg.*	*MAD*	*avg.*	*MAD*	*avg.*	*MAD*
Arith	17,161	93.5	17,264	**2.9**	21,532	120.6	11,346	**0.3**
BitVec	29,141	41.0	29,247	**14.6**	34,245	121.4	28,248	**14.6**
Equality	101,912	286.0	101,778	**4.0**	107,718	108.8	107,885	**10.7**
Equality+LinearArith	77,933	208.4	77,620	**32.8**	72,603	254.5	73,041	**22.2**
Equality+MachineArith	91,935	23.4	91,518	**6.8**	74,037	452.8	73,539	**37.2**
Equality+NonLinearArith	97,462	351.0	95,961	**10.8**	89,615	375.5	90,551	**17.1**
FPArith	18,089	49.6	17,920	**49.5**	14,371	143.2	14,214	**28.5**
QF_Bitvec	149,591	282.8	153,082	**208.9**	104,282	751.8	94,428	**174.6**
QF_Equality	2,408	49.0	2,435	**3.7**	1,285	64.4	1,195	**1.0**
QF_Equality+Bitvec	40,712	157.1	40,246	**153.7**	36,282	265.5	35,870	**84.7**
QF_Equality+LinearArith	14,927	46.4	16,070	**44.9**	8,278	296.4	4,862	**179.5**
QF_Equality+NonLinearArith	34,344	422.9	34,234	**276.0**	25,811	374.6	25,731	**175.3**
QF_FPArith	28,971	289.6	28,063	**108.1**	56,349	324.4	55,262	**106.3**
QF_LinearIntArith	242,429	402.8	244,127	**148.7**	151,935	606.8	132,783	**345.7**
QF_LinearRealArith	30,567	313.6	29,539	**101.0**	23,547	310.8	21,260	**160.4**
QF_NonLinearIntArith	34,201	615.7	34,267	**26.3**	23,914	194.7	22,265	**6.8**
QF_NonLinearRealArith	39,951	86.8	39,377	**27.9**	26,186	188.3	28,016	**131.7**
QF_Strings	34,092	341.4	33,255	**121.6**	56,970	321.4	56,420	**158.4**
Family (*mariposa*)								
Dice^*_F	–	–	–	–	7,531	559.3	8,951	**545.3**
Komodo_D	–	–	–	–	12,561	264.2	13,327	**59.9**
Komodo_S	–	–	–	–	3,284	79.9	3,181	**17.2**
VeriBetrKV_D	–	–	–	–	12,623	414.4	12,612	**169.0**
VeriBetrKV_L	–	–	–	–	24,889	203.6	23,971	**172.7**
vWasm_F	–	–	–	–	2,919	9.5	3,018	**0.2**

TABLE II: PR-2 and MAD scores on mutated benchmarks before and after normalization.

We use penalized runtime (PR-2) and Median Absolute Deviation (MAD) as metrics. Penalized runtime the sum of the time taken for solved benchmarks plus a penalty equal to two times the timeout for each unsolved benchmark (timeouts, memory outs, or other errors). PR-2 thus combines elements of both total time and number of solved benchmarks into a single metric. MAD measures how much variation there is among a set of results, with lower numbers indicating less variation and higher numbers indicating more variation. Applying MAD to PR-2 scores is thus a good proxy for stability.

Results are shown in Table II for each division (in *smtlib*) and family (in *mariposa*). We report results on the benchmarks before normalization (*no norm.*) and after normalization with super-patterns (*norm.*). The column labeled *avg* contains the average PR-2 score, computed as follows. Recall that we create ten mutations for each benchmark. Each mutation is based on a specific random seed. We compute the total PR-2 score for all the benchmarks for each seed separately. We then take the average of these ten PR-2 scores. The MAD is also computed over these ten cumulative PR-2 scores (one for each seed).

We see that, for both solvers, the performance (avg. column) on normalized benchmarks is generally comparable with that of non-normalized benchmarks, showing that, on average, normalization does not appear to greatly affect runtime.

More importantly, the MAD score improves significantly in *all* cases, sometimes by more than an order of magnitude (e.g., in the Equality division). This strongly suggests that our normalization algorithm improves stability. Even for benchmarks for which the normalizer completely fails to produce normal forms (e.g., the *mariposa* benchmarks that always still have 10 distinct benchmarks after normalization), the stability improves, sometimes significantly. This provides promising evidence that full normalization is not required for improving stability. An improvement in similarity may be sufficient.

C. Code Optimizations

Some of the benchmarks in our benchmark sets are extremely large, with sizes up to hundreds of megabytes and with up to hundreds of thousands of user-defined symbols and assertions. In this section, we discuss three optimizations that are crucial for scalability of our algorithm on such benchmarks.

a) Pattern Compression: As presented in Section III, formulas are sequences whose size corresponds to the size of their abstract syntax tree. However, internally, SMT solvers represent formulas as directed acyclic graphs (DAGs), because they often share subterms. The DAG representation can be exponentially more concise. It is thus essential to also incorporate this kind of sharing in our representation of patterns. In our implementation, we represent a pattern as an index into a dictionary of the subterms in the tree representation of the pattern. For example, the formula $\langle +, *, x, x, *, x, x\rangle$ would be represented as the index $\mathbf{2}$ into the dictionary $(\mathbf{1} : \langle *, x, x\rangle, \mathbf{2} : \langle +, \mathbf{1}, \mathbf{1}\rangle)$.

b) Super-pattern Computation: A naive way to compute the super-pattern for a symbol s is to traverse all of the assertions and look up the role of s in every assertion. However, this approach is not scalable for large benchmarks. To address this issue, we perform a one-time indexing pass over the assertions to create arrays for each symbol s containing pointers to only the assertions in which s appears. This way, we only need to traverse these assertions when calculating the super-pattern for s. This optimization reduces the number of lookups from hundreds of millions to hundreds of thousands on some problematic benchmarks, improving the normalization time by more than an order of magnitude.

c) Super-pattern Compression: This optimization leverages the sparsity of super-patterns: in large benchmarks, each symbol typically appears in only a small subset of the assertions. Thus, the role of a symbol is likely to be 0 in most assertions. To exploit this, we represent super-patterns as sequences consisting of the non-zero role values interleaved with counts of the number of zeros.

VI. Conclusion

Our normalization algorithm is a promising step towards a more stable and predictable SMT solving experience. It generally scales well and is applicable to a wide range of benchmarks and logics. We have shown that it can often produce a single unique normal form for a set of mutated benchmarks, and even when it cannot, it greatly increases the similarity of the benchmarks. We also saw that, on average, it improves stability without significantly affecting performance.

In future work, we intend to expand our normalizer to handle other mutations such as antisymmetric operator replacement and operand swapping for commutative operators. We also plan to explore whether additional normalization techniques can be used to further improve our results. Finally, we plan to investigate the use of our normalization algorithm as a preprocessing step to improve the hit rate of applications that cache formulas in order to reuse solving results.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

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

[3] Clark Barrett and Cesare Tinelli. POSE: Phase ii: An open-source ecosystem for the cvc5 SMT solver, 2023.

[4] Nikolaj S. Bjørner. SMT solvers: Foundations and applications. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 24–32. IOS Press, 2016.

[5] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using webassembly. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 1975–1992. USENIX Association, 2022.

[6] Martin Bromberger, François Bobot, and Martin Jonáš. SMT Competition (SMT-COMP) 2024, 2024.

[7] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[8] Mike Dodds. Formally verifying industry cryptography. *IEEE Secur. Priv.*, 20(3):65–70, 2022.

[9] Herbert Enderton and Herbert B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[10] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305. ACM, 2017.

[11] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 99–115. USENIX Association, 2020.

[12] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 165–181. USENIX Association, 2014.

[13] Robert Jones. Private communication, 2024.

[14] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[15] Misha Lavrov. Algorithm for finding a "normal" form for a set of strings under character-mapping isomorphism? Mathematics Stack Exchange. https://math.stackexchange.com/q/2925668 (version: 2018-09-21).

[16] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[17] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016.

[18] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–28, 2022.

[19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019.

[20] Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. Smt-lib release 2024 (non-incremental benchmarks), April 2024.

[21] Neha Rungta. A billion SMT queries a day (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022.

[22] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[23] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. Dice*: A formally verified implementation of DICE measured boot. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1091–1107. USENIX Association, 2021.

[24] Tjark Weber. Scrambling and descrambling SMT-LIB benchmarks. In Tim King and Ruzica Piskac, editors, *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories, Coimbra, Portugal, July 1-2, 2016*, volume 1617 of *CEUR Workshop Proceedings*, pages 31–40, July 2016.

[25] Y. Zhou, J. Bosamiya, J. G. Li, M. J. H. Heule, and B. Parno. Context pruning for more robust smt-based program verification. In N. Narodytska and P. Rümmer, editors, *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*, pages 59–69. TU Wien Academic Press, 2024.

[26] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 178–188. IEEE, 2023.

[27] Yi Zhou, Bryan Parno, Marijn Heule, and Jay Bosamiya. Mariposa: Measuring smt instability in automated program verification [benchmarks], January 2025.

VII. Appendix

Lemma VII.1. *Let A be a sequence of formulas. If $A_R = R(A_P)$, where $A_P \in P(C(S(A)))$ and R is a renaming, then $P(C(S(A_R))) = P(C(S(A)))$.*

Proof. First, note that for any formula sequence A, we have $R(P(C(S(A)))) = C(S(R(P(A))))$. This is because the first generates a set of equivalent formula sequences and then renames the symbols, while the second renames the symbols and then generates a set of equivalent formula sequences, but these two transformations are indpendent, so the result is the same, regardless of which order they are done in.

Now, we can write $A_R = R(P(A'))$ for some $A' \in C(S(A))$. It follows that $P(C(S(A_R))) = P(C(S(R(P(A'))))) = P(R(P(C(S(A'))))) = P(C(S(A')))$, since computing the pattern of a renaming of a pattern is just the same as computing the pattern. It remains to show that $P(C(S(A'))) = P(C(S(A)))$.

But $A' \in C(S(A))$, which means that A can be obtained from A' by swapping zero or more commutative operators and permuting the order of the formulas in A'. But then, any element of $C(S(A))$ can be obtained from A' by swapping commutative operators and permuting the order, so $C(S(A'))$ is just the same as $C(S(A))$. Thus, $P(C(S(A'))) = P(C(S(A)))$. □

Lemma VII.2. *For any formula A, $P(C(S(A)))$ has a unique minimal element, according to $\prec$.*

Proof. When comparing two elements of $P(C(S(A)))$, each of which is a sequence of patterns, we concatenate the patterns together and compare them with $\prec$. Since $\prec$ is a total order, one of them is always smaller. Thus there is a minimal element of $P(C(S(A)))$. □

Theorem III.8. *Function $\mathcal{N}$ is normalizing.*

Proof. Notice that $\mathcal{N}(A) = \mathbb{R}(A')$, where $\mathbb{R}$ is the identity function and A' is the $\prec$-minimal element of $P(C(S(A)))$. It is not hard to see that $\mathbb{R}$ is also a renaming, so the first requirement is met. Now, let $M_1 = R_1(M_1')$ and $M_2 = R_2(M_2')$, with $M_1', M_2' \in P(C(S(A)))$, where R_1, R_2 are renamings. By Lemma VII.1, $P(C(S(M_1))) = P(C(S(A))) = P(C(S(M_2)))$. But $P(C(S(A)))$ has a unique minimal element, A' by Lemma VII.2. Thus, $\mathcal{N}(M_1) = A' = \mathcal{N}(M_2)$. □

Theorem III.9. *Let N be a normalizing function. Then, computing $N(A)$ for an arbitrary A is as hard as solving graph isomorphism.*

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two undirected graphs. Assume without loss of generality that $V_1 \cap V_2 = \emptyset$. For each $v \in \{V_1 \cup V_2\}$, let $u(v)$ map v to some unique user-defined symbol (i.e., u is injective). Now, define $A_i = \{\langle f, x\rangle \mid x \in V_i\} \cup \{\langle =, x, y\rangle \mid \exists (v, v') \in E_i.\ \{x, y\} = \{u(v), u(v')\}\}$, where f is some Boolean predicate. Note that G_i can be recovered from A_i, simply by creating a vertex for every user-defined symbol appearing as an argument of f in A_i and then adding an edge between $u^{-1}(x)$ and $u^{-1}(y)$ whenever the formula $\langle =, x, y\rangle$ appears in A_i. Furthermore, shuffling the formulas in A_i, permuting the order of the operands in equalities (the only commutative operator appearing in A_i), or renaming the user-defined symbols does not change the structure of the graph being represented. Thus, all elements of $P(C(S(A_i)))$ represent isomorphic graphs.

Now, suppose $N(A_1) = N(A_2)$, For $i \in [1, 2]$, we know that $N(A_i) = R(A_i')$ for some $A_i' \in P(C(S(A_i)))$. Furthermore, because renamings are injective, $A_1' = P(R(A_1)) = P(N(A_1)) = P(N(A_2)) = P(R(A_2)) = A_2'$. But for $i \in [1, 2]$, the graph represented by A_i is isomorphic to the graph represented by A_i'. Thus the graph represented by A_1, namely G_1, is isomorphic to the graph represented by A_2, which is G_2.

On the other hand, suppose G_1 and G_2 are isomorphic. Let $h : V_1 \to V_2$ be the isomorphism function for the graph vertices. Let R be a renaming such that $\forall\, v.\, R(P(v)) = h(v)$. Applying this renaming to A_1 must be equivalent (modulo order) to A_2. In other words, $A_1 \in R(P(S(A_2)))$. Then, because N is normalizing, we must have $N(A_1) = N(A_2)$. Thus, computing N is at least as difficult as graph isomorphism. □