A Conjecture Regarding SMT Instability

Can Cebeci[1,*], Nikolaj Bjørner[2], George Candea[1] and Clément Pit-Claudel[1]

[1]EPFL, Lausanne, Switzerland
[2]Microsoft Research, Redmond, USA

Abstract

Automated verifiers rely on SMT solvers as a backend for proof automation. This simplifies verification in the common case, but also creates usability challenges due to instability. Previous work proposes mitigation strategies motivated by the assumption that instability is a *fundamental* theoretical limitation. We conjecture that the instability experienced by verifiers today is often caused by *fixable* engineering problems, and is thus not fundamental. Should this be true, the more consequential approach to addressing instability would be to identify root causes and connect them with solver improvements.

We conducted case studies on 11 deemed-unstable queries from existing verification projects and from issues reported on the Z3 repository, with the goal of diagnosing the root causes of instability. For all of them, instability is attributable to solver bugs, misconfiguration, or misaligned expectations. We present our analysis and draw conclusions regarding better instability metrics, interfaces that make instability explicit, systematic debugging methods, and easier-to-debug SMT solvers.

Keywords

instability, debugging

1. Introduction

Formal verification often relies on SMT solvers as a backend for proof automation. In the common case, SMT solvers significantly lower the amount of effort and expertise required for verification, as they discharge most proofs quickly.

Yet, SMT-based verifiers face significant usability challenges. A major reason is that SMT solvers suffer from instability, whereby minor changes to the input query, such as addition or removal of redundant constraints, can have disproportionate effects on performance, or even the outcome (e.g. turn an `unsat` into an `unknown`). Recent work has shown that even purely syntactic mutations such as renaming variables or reordering assertions can have such effects [1]. This makes verifiers unpredictable and often frustrating to use [2, 3, 4].

Existing (and ongoing) work [5, 6, 7] proposes mitigations for instability issues but does not analyze their root causes, so it is unclear *why* the mitigations work, or whether they generalize. Accordingly, the state of the art in building SMT-based verifiers is based on intuition: experts know empirically that some features do not work well together (e.g., quantifier-heavy encodings and model-based quantifier instantiation [8]), and that some features are best avoided (e.g., non-linear arithmetic). Overall, experts perceive a fundamental tradeoff: that instability needs to be addressed at the cost of either the expressiveness of the encoding (by using fewer solver features), or proof automation (by reducing the context visible to the solver, so that there is less room for "bad luck"). We are not convinced that this tradeoff is inherent to using SMT solvers.

We formulate a conjecture that the instability experienced by verifiers today is mostly caused by *fixable* engineering problems, not *fundamental* theoretical limitations. SMT is combinatorially hard, so solvers must rely on heuristics to be efficient: in that sense, instability may be inescapable. However, we speculate that, in the context of a finite set of existing tools and their encodings, the heuristics that already work well (albeit sometimes unstably) can be made stable through dedicated engineering.

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK
*Corresponding author.
can.cebeci@epfl.ch (C. Cebeci); nbjorner@microsoft.com (N. Bjørner); george.candea@epfl.ch (G. Candea); clement.pit-claudel@epfl.ch (C. Pit-Claudel)

Motivated by this conjecture, we set out to investigate the root causes of instability. The SMT community does not have the tools and techniques today to pinpoint root causes at scale, generically, and automatically. In lieu of this and as a starting point, we manually examined a handful of queries in detail. We conducted case studies on 11 queries that were unstable with Z3 [9], taken from existing verification projects [10, 2, 11] and the Z3 forum.

In all 11 case studies, we found that the heart of the problem was neither a bad encoding nor bad luck. Rather, they boiled down to concrete, fixable root causes:

- 6 of the queries were unstable due to **bugs in the solver** (§4.1). We have implemented three bug fixes addressing all 6 queries, which have been merged upstream.
- 2 were unstable due to **misconfiguration** (§4.2). These are each stabilized by a solver parameter that is set through the command line. In both cases, the parameter's effect is intuitive and its necessity is obvious in hindsight.
- 3 were unstable due to **misasligned expectations** (§4.3), specifically relating to how triggers are treated in the presence of relevancy filtering [12], which behaves according to its specification but still violates user expectations. These queries are stabilized by rewriting the triggers.

In the following sections, we present our case studies and make three calls to action:

- *Tools and methods for debugging instability.* Our case studies were costly: a few months of training plus multiple person-days per query. Some of our work can be systematized and automated, so as to make instability debugging accessible and cost-feasible. Such tool support would not only benefit users, but also the authors of SMT solvers, who stand to reap significant benefits when addressing user issues or during development.
- *Better instability metrics.* Existing metrics [1, 13] are timeout-dependent and sensitive to variability in solving time, even when the variability is expected. This may both lead to stable queries being categorized as unstable (if the timeout is too short) and vice versa (if the timeout is too long). We need a metric that goes beyond simple statistical measures and captures instability as experienced by users (e.g., bimodality of time to solve).
- *Controlled randomness in SMT solvers.* Modern solvers are affected by many sources of randomness (e.g., variable and function names, assertion order). With current verifiers, exploring the space of randomized choices is nearly impossible. To remedy this, we need to refactor solvers such that all randomized choices are determined explicitly by user-controlled random seeds. This would be a step towards making instability easier to identify and solvers easier to debug.

2. Background

In this section we define instability, and discuss recent efforts towards mitigating it. Then, we present an overview of a CDCL(T)-based SMT solver's architecture and operation, highlighting the aspects most relevant to our discussion. We focus specifically on Z3, as it is the solver we used in our case studies.

2.1. Defining, measuring, and addressing instability

Proof instability, also called brittleness [14], the butterfly effect [15] and solver explosion [10], is a commonly experienced and acknowledged problem with SMT-based verifiers. While there is no universally agreed-upon definition, in this paper we use "instability" to broadly refer to solver behavior that is unpredictable and inconsistent from the user's perspective. This includes disproportionate performance variations caused by minor changes to input queries, proofs breaking after solver or verifier upgrades, as well as sensitivity to redundant constraints, seemingly trivial simplifications, or reseeding. Existing work quantifies instability by fuzzing queries with syntactic mutations [1] and random seeds [13], and measuring variations in performance or outcome. These techniques detect a

wide range of unstable queries, but not all. As shown in the rest of this paper, **they are not perfect metrics, and certain kinds of instability are out of their reach.**

Recent work has proposed various measures to reduce instability. One general direction is to pre-process queries, to simplify [16], prune [6], or canonicalize [5] them. Another is to restrict the verifier's SMT encoding to simpler theories [10, 17, 18, 19], often at the cost of limiting expressivity or automation. Some verifiers offer language features that let users control the encoding for specific code sections [7, 20], confining the use of theories and axioms believed to cause instability. A common thread to these techniques is that they aim to mitigate rather than fix instability. **The mitigations have empirically been shown to work, but it is not clear whether they address root causes or whether the tradeoffs they impose are strictly necessary.**

2.2. The architecture and operation of a CDCL(T)-based solver

This section presents an overview of Z3's architecture and operation, using the following query as a running example:

$$(x < 0) \wedge (y < 0) \wedge ((x + y > 0) \vee (x \cdot y < 0))$$

Z3 comprises a ***CDCL(T) core***, and a set of ***theories***. The former is essentially a SAT solver that treats theory atoms (e.g., $(x < 0)$, $(x + y > 0)$) as independent boolean variables and aims to construct a boolean model that satisfies the propositional constraints (e.g., $\{(x < 0) :- \text{true}, (y < 0) :- \text{true}, (x + y > 0) :- \text{false}, (x \cdot y < 0) :- \text{true}\}$). Theories are mainly responsible for ensuring that the set of propositional assignments in the model is consistent with respect to theory semantics. For this model, the theory of arithmetic would raise a conflict: $(x < 0) :- \text{true}$, $(y < 0) :- \text{true}$, and $(x \cdot y < 0) :- \text{true}$ cannot simultaneously hold.

The CDCL(T) core makes ***decisions*** on propositional assignments one by one, either through boolean constraint propagation (BCP) or by ***branching***. After each round of decisions, it performs ***theory propagation*** to see if the current (partial) model is consistent with the theories. In this example, before any branching, it decides that any model must have $(x < 0) :- \text{true}$ and $(y < 0) :- \text{true}$ through BCP, as these appear as unit clauses. Then, it propagates both assignments to the theory of arithmetic, which says that the model is consistent. When there is no unit clause left to be propagated, Z3 branches on a decision, creating a ***backtracking*** point. Let's assume that it first decides to explore the case where $(x + y > 0) := \text{true}$. As soon as this decision is theory-propagated, the theory of arithmetic raises a conflict. In response, the CDCL(T) core backtracks to the previous partial model, and augments it with $(x + y > 0) := \text{false}$. Finally, the CDCL(T) core decides $(x \cdot y < 0) :- \text{true}$ (through BCP), which causes another arithmetic conflict. As the CDCL(T) core cannot backtrack anymore, it decides that the query is `unsat`.

Z3 refines the basic CDCL(T) architecture with many heuristics and optimizations. One that is particularly important in the context of this paper is ***relevancy filtering*** [12]: in order to minimize the load on theories, the CDCL(T) core theory-propagates only a subset of the current assignments, those it deems relevant to the current branch of the proof. For instance, in the last round of theory propagation in our example, $(x + y > 0) := \text{false}$ would be filtered out, as $(x + y > 0)$ only appears in the query within a disjunction, and the fact that it has been assigned to false has no consequence on the truth value of the disjunction.

Another important mechanism to highlight is ***trigger-based quantifier instantiation***, also called E-matching. We do not present E-matching here, and instead refer to prior literature [15]. We do, however, note that the CDCL(T) core treats the E-matching engine like any theory in the context of relevancy filtering: it keeps ground terms that only appear in irrelevant atoms from matching any triggers. Previous work [15] has explored the contribution of matching loops to instability. We do not discuss matching loops further, as they did not turn out to be the root cause for any of our case studies.

3. Experimental Setup

We have conducted case studies over 11 unstable queries. 4 of these were produced by early versions of TPot [10]; their instability motivated some of TPot's encoding choices and optimizations. 4 of the queries come from Z3 bug reports or private email exchanges with verifier developers. These include queries produced by Dafny [21], Ivy [22] and Verus [7]. The remaining 3 queries are taken from the Mariposa benchmark [23].

We did not choose the queries for our case studies systematically. Instead, we started with examples we organically came across and were puzzled by. Then, we investigated issues that were recently raised with Z3 developers or directly shared with us, as well as some of the simpler queries in the Mariposa benchmark.

We used Z3 builds between versions 4.13.4 and 4.14.2 for all of our case studies.

4. Findings

Origin	Query name	Issue	Solution
TPot	raw [24]	solver bug	`patch-dynack` [25]
TPot	april2nd [26]	solver bug	`patch-fixedeq` [27]
TPot	offset [28]	solver bug	`patch-fixedeq`
TPot	bv_rewrite [29]	misconfiguration	`tactic.default_tactic=smt`
Forum/Dafny	issue_7444 [30]	wrong expectations	rewrite triggers
Forum/Dafny	IfNorm0	wrong expectations	remove extraneous quantifier
Forum/Verus	root0	wrong expectations	rewrite triggers
Forum/Ivy	thing2 [31]	solver bug	`patch-mbqi` [32]
Mariposa [23]	set_bit_to_0_self_uint64	solver bug	`patch-dynack`
Mariposa	set_bit_to_1_self_uint64	solver bug	`patch-dynack`
Mariposa	lemma_2toX	misconfiguration	`smt.qi.eager_threshold=200`

Table 1
Summary of our cases studies. The solutions for `bv_rewrite` and `lemma_2toX` are command line arguments to Z3 that set the relevant parameters. `patch-dynack`, `patch-fixedeq` and `patch-mbqi` are bugfixes explained in §4.1.

Table 1 summarizes our case studies. All 11 queries we investigated turned out to be unstable due to fixable root causes: 6 due to solver bugs, 2 due to misconfiguration, and 3 due to misaligned expectations.

We also stability-tested all 11 queries with Mariposa, using for each query the solver version instability was originally reported with. With its default configuration (using a 60-second timeout), Mariposa identifies 3 of the queries as unstable (`thing2`, `IfNorm0`, `bv_rewrite`), in addition to the three that were already taken from the Mariposa benchmark. It judges the other five queries to be unsolvable. For three of these (`raw`, `april2nd`, `offset`), all Mariposa mutants time out at 60 seconds with the original solver version, and are solved in less than a second after Z3 bugfixes. For the remaining two (`root0`, `issue_7444`), all mutants terminate fast and yield `unknown`. All five queries are in fact unstable according to our informal definition in §2.1, as they can be made to succeed with minor modifications that are semantically irrelevant.

In the rest of this section, we present our findings and insights.

4.1. Instability may be caused by solver bugs

SMT solvers, like all software, have bugs. It is well-known [33] that these bugs may lead to unsoundness; It is not as well-documented that they may lead to instability as well.

We have identified and fixed three bugs that lead to instability in Z3. All three patches have been merged upstream.

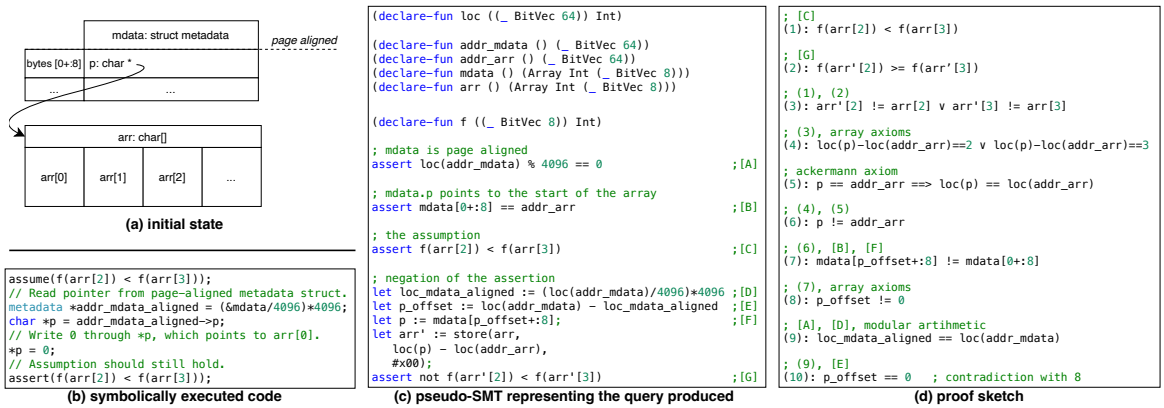

Figure 1: A simplified version of the `raw` query, which was stabilized by `patch-dynack`. (a) and (b) explain the context for the query: given the initial state (a) for a C program, the assumption in (b) should imply the assertion. TPot encodes this into the query described in (c). For simplicity, we present the query in pseudo-SMT, where `mdata[i+:n]` denotes a nested bitvector concatenation of depth `n`, consisting of byte-long selections from `mdata` starting at index `i` (i.e., `concat(mdata[i+n-1], concat(mdata[i+n-2], ...))`. `arr[i]` similarly denotes a `select(arr, i)`. (d) presents an example sketch of an unsatisfiability proof that one could expect the solver to discover.

- `patch-dynack` fixes a bug that kept dynamic Ackermannization axioms (see §4.1.1) from being propagated to theories properly and that caused the relevancy of certain atoms to be forgotten during backtracking.
- `patch-fixedeq` fixes relevancy tracking for equality lemmas between bitvectors that are assigned the same value by the bitvector theory, such that the equality is propagated to the egraph structure.
- `patch-mbqi` fixes a bug related to model-based quantifier instantiation for uninterpreted sorts with finite domains, whereby Z3 would complete partial functions models with arbitrary values.

`patch-dynack` and `patch-fixedeq` are bugs that affect the communication between the CDCL(T) core and the theories. This is a channel that is regulated by relevancy filtering, and can easily keep the solver from finding the intended proof. Section §4.1.1 presents `patch-dynack` in detail.

4.1.1. Deep dive on a solver bug

We discovered the bug related to `patch-dynack` while experimenting with an early version of TPot, when the solver got stuck on the `raw` query. This was an instance of instability: `raw` mostly comprised the same assertions as previous queries TPot made on the same symbolic execution path, and used the same logic (quantifier-free bitvectors and arrays), so we expected Z3 to discharge the query with similar reasoning.

Fig. 1 presents a simplified (and still unstable) version of the query, along with the context that produced it and the sketch of an intended proof. The original query included 86 assertions spanning 3600 lines of SMT2. The proof sketch justifies why we expect the query to be easily solvable, and helps explain how the bug leads to explosion. We have found writing pen-and-paper proof sketches to be helpful for debugging as well.

Dynamic Ackermannization [34] is a technique used by Z3 to significantly speed up reasoning about transitivity of equality and congruence. In the intended proof, the Ackermann axiom in step 5 is essential. Without it, the solver would have to discover the same fact through congruence closure, which results in producing much weaker lemmas [35].

While Ackermannization itself worked as expected for the query, a Z3 bug related to the interaction between Ackermannization and relevancy filtering kept step (6) from proceeding: Ackermannization would create fresh literals for step (5) (e.g., `p == addr_arr`) without marking them as relevant. This effectively made the Ackermann clauses invisible to the theories, so the theory of arithmetic would not

learn that assuming `p == addr_arr` propagates `loc(p) == loc(addr_arr)`. Consequently, it would not learn that `p != addr_arr`.

When Z3 backtracks it will retain certain axioms (i.e., theory lemmas, including Ackermann axioms) but forget about their relevancy. So the lifetime of the lemma is not scope-bound, but its relevancy is. This essentially results in the axioms only participating in propositional propagation (within the CDCL(T) core), but not theory propagation. This behavior is intended if the solver creates theory axioms using existing literals, but works against leveraging theory axioms with fresh literals across backtracking points.

4.2. Instability may be caused by misconfiguration

Solvers come with many configurable parameters that toggle heuristics, determine thresholds, set resource limits, and so on. The current version of Z3 (4.14.2) has 657 parameters configurable through the command line. Intuitively, the configuration should play a role in whether the solver behaves as expected consistently for an encoding scheme; indeed, two of our case studies turned out to be configuration issues.

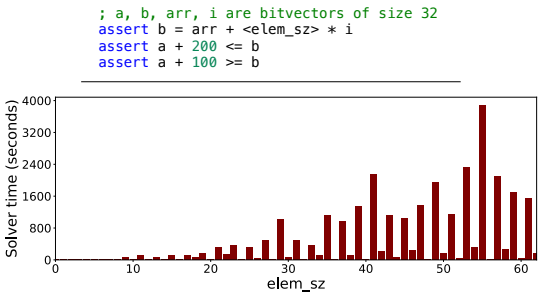

Figure 2: Pseudo-SMT describing `bv_rewrite`, and the performance impact of `elem_sz`.

```
lemma lemma_2toX()
    ensures power2(64) == 18446744073709551616;
    ensures power2(60) == 1152921504606846976;
    ensures power2(32) == 4294967296;
    ensures power2(24) == 16777216;
    ensures power2(19) == 524288;
    ensures power2(16) == 65536;
    ensures power2(8) ==  256;
{
    reveal_power2();
}
```

Figure 3: The Dafny function that gets encoded into the `lemma_2toX` query.

4.2.1. A misconfigured heuristic

`bv_rewrite.smt2` is a minimized (to 14 lines) version of an unstable query that was produced by TPot. It involves 4 variables, all of which are bitvectors of size 32. As shown in Fig. 2, the time it takes Z3 to solve `bv_rewrite.smt2` depends heavily on a bitvector coefficient (`elem_sz`) appearing only in an assertion that is redundant and therefore should be inconsequential.

With the default configuration, Z3 determines that the query only involves quantifier-free constraints over bitvectors, and so it is essentially a boolean satisfiability problem. Then Z3 decides to bit-blast the query, representing each bit with a boolean variable, and discharge it using the built-in SAT solver (which is separate from the CDCL(T) core), without ever involving the actual SMT solver. Using the command line argument `tactic.default_tactic=smt` prevents this from happening and makes Z3 terminate instantly irrespective of the coefficient.

4.2.2. An inappropriate resource limit

`lemma_2toX.smt2` is a query produced by Dafny to prove the lemma in Fig. 3, as a part of the IronFleet [36] project. The query is unstable with variable outcome: depending on the random seed, Z3 may quickly return either `unsat` or `unknown`.

In the SMT encoding, Dafny represents `power2` with an uninterpreted function, and provides the usual [37] recursive definition ($\forall e.\text{power2}(e) = \text{ITE}(e = 0, 1, 2 \cdot \text{power2}(e-1))$) as a quantified axiom. For each `ensures` statement in Fig. 3, Dafny expects Z3 to unfold the recursive definition n times to compute $\text{power2}(n)$.

The problem is that the global threshold Dafny sets on quantifier instantiation depth (`smt.qi.eager_threshold=100`) is too small to fully unfold $\mathrm{power2}(64)$. We have empirically observed using SMTScope [38], the latest version of the Axiom Profiler [39], that it does not allow $\mathrm{power2}$ to be unrolled more than 32 times. So, Z3 can not close the branches of the proof involving $\mathrm{power2}(64)$ and $\mathrm{power2}(60)$.

The misconfiguration here is obvious in hindsight. In practice, it was not easily noticable as it made the query unstable, not consistently `unknown`. This is because when we get lucky, Z3 reuses unfoldings that were used to close branches corresponding to the last five `ensures` to close the first two as well. So whether Z3 returns `unsat` or `unknown` depends on the order in which it explores different branches. Setting the threshold to 200 allows all branches to close in isolation and stabilizes the query.

One may also configure Z3 such that quantifier instances are effectively shared across branches, by disabling relevancy filtering (`smt.relevancy=0`). Doing so indeed stabilizes this particular query, but makes most other Dafny queries to explode, due to Dafny's quantifer-heavy encoding.

4.3. Instability may be caused by misaligned expectations

Developers of SMT-based verifiers treat solver optimizations as black boxes that are only relevant as far as performance is concerned. They assume that, barring timeouts and resource limits, toggling an optimization may change the solver's performance but not its completeness (e.g., may not turn an `unsat` into an `unknown`), otherwise the solver must have a bug. This assumption is incorrect, at least in the case of relevancy filtering combined with trigger-based quantifier instantiation.

Our case studies include two queries with unstable outcomes for which the root cause is a mismatch between how verifier developers expect triggers to be treated and how Z3 treats them in the presence of relevancy filtering. These are not Z3 bugs; relevancy filtering behaves according to how it is specified, through semantic tableau rules, even though it violates user expectations. In §4.3.1 we present a minimal reproduction and in-depth discussion on this mismatch.

```
1  (declare-fun A (Int) Bool)
2  (declare-fun B (Int) Bool)
3
4  (assert (or (A 0) (B 0)))
5  (assert (forall ((x Int))
6      (! (and (not (A x)) (not (B x)))
7          :pattern ((A x)))
8  ))
9  (check-sat)
```

Figure 4: Minimal SMT query reproducing the instability caused by the expectation mismatch.

4.3.1. A minimal example

Consider the query in Fig. 4. The query is clearly `unsat`, as instantiating the quantifier with $x :- 0$ trivially contradicts the first assertion. The conventional understanding of triggers, as described by Leino and Pit-Claudel [15], dictates that the ground term `(A 0)` would match the trigger and produce the expected instance, and Z3 would return `unsat`. Surprisingly, in the presence of relevancy filtering, Z3 may return `unknown` depending on the branching order, precisely 50% of the time. We explain this frequency in §4.3.2.

The core of the problem is that there is a branch of the proof that does not close itself; it depends on the quantifier instance but does not trigger instantiation. In the branch where $A(0)$ is false and $B(0)$ is true, Z3 considers $A(0)$ to be an irrelevant disjunct in `(or (A 0) (B 0))`, as its truth value does not affect the value of the disjunction. Therefore, Z3 keeps the assignment $A(0) :- \mathrm{false}$ from being propagated to any theory, including the E-matching engine. So, the branch only closes if Z3 visits one of the branches that triggers instantiation (e.g., $A(0) :- \mathrm{true}, B(0) :- \mathrm{false}$) before this one and reuses the instance.

Another way to understand the problem is to realize that relevancy filtering effectively simulates tableau-search, whereby the solver structurally branches on the problem. In this example, the solver looks at the first assertion and branches on satisfying $A(0)$ versus $B(0)$. The latter branch is essentially equivalent to solving a version of the query with $A(0)$ omitted from the first assertion, and that version *should* be unknown according to Leino and Pit-Claudel's description.

4.3.2. Quantitatively understanding the minimal example

The query's success depends purely on the branching order and 50% of branching orders lead to success. We experimented with 8 variants of the query, changing the polarities of $A(0)$ and $B(0)$, and toggling the pattern between $A(x)$ and $B(x)$.

Some of these seemed to be consistently `unsat` or `unknown` irrespective of the random seed, suggesting that these variations play a role in the query's success. On closer look, some variations may fix a branching order with Z3's default configuration: Z3 will always guess false for the first atom it branches on irrespectively of the random seed (the actual guessing process - phase caching - is somewhat more involved, but the caches are initialized to false). Also, the way Z3 chooses the first atom to branch on involves randomness but it is not uniformly random. To work around these issues, we made a surgical modification on Z3's source code to set the frequency of randomized case splits to 1.00 (the default is 0.01). This frequency is hard-coded and is not configurable through the command line. We then ran Z3 with `smt.phase_selection=5` (random).

Assignment 1			Assignment 2			Result	Probability
branch?	literal	relevant?	branch?	literal	relevant?		
yes	**A(0)**	**yes**				unsat	1/4
	¬A(0)	no	no	B(0)	yes	unknown	1/4
	B(0)	yes	yes	A(0)	no	unknown	1/8
				¬A(0)	no	unknown	1/8
	¬B(0)	no	no	**A(0)**	**yes**	unsat	1/4
					Total:	unsat	1/2
						unknown	1/2

Table 2
All five possible branching orders for the minimal example in Fig. 4, along with their respective outcomes and probabilities. A(0) represents the assignment of the atom A(0) to true, and likewise ¬A(0) represents its assignment to false. When A(0) is assigned a truth value and is in a relevant position, the quantifier is instantiated and Z3 returns `unsat` after a round of boolean constraint propagation, without further branching or assignments. Otherwise, when a branch assigns a truth value to both A(0) and B(0), Z3 returns unknown since it has constructed a complete model without finding any conflicts. Since we have configured Z3 to make branching decisions uniformly, each time Z3 branches, the probability of each branch is divided by the number of options.

Table 2 presents all possible branching orders, explains the outcomes they lead to, and explains the 50% success rate. We validated this explanation by instrumenting Z3 to log all decisions and counting the number of runs following each branching order.

5. Discussion

In this section, we discuss our experience with the case studies. We hope the lessons we learned will inform future work on addressing instability.

5.1. Sources of randomness

In §4.3.2, we describe how certain variations on the query in Fig. 4 seem to have implications on stability, when in fact all they do is resample a random distribution that is unaffected by the random seed. For this query, the order of disjuncts, and the symmetric choice of the trigger term between the two disjuncts, are sources of randomness. This is an example of how having multiple sources of randomness can lead to confusion while identifying and debugging instability.

Another such example is the `IfNorm0` query. The original query produced by Dafny succeeds with all random seeds, but asking Dafny to normalize the names of functions and variables (through a Dafny command line option) makes the query explode with all random seeds. This pattern is confusing, as it suggests that normalization is to blame for instability. The pattern occurs because Z3 reorders disjunctions using a name-dependent order and Dafny sets the `smt.case_split=3` parameter, which orders case splits structurally, starting from the leftmost disjunct. This makes the case split order insensitive to the random seed, but sensitive to variable and function names.

Ensuring that all randomness is controllable through an explicit random seed would be a step towards making instability easier to identify and address.

5.2. Examining, comparing, and replaying solver traces

Existing approaches to examining Z3 traces [39] are based on statistical measures (e.g., number of instances of each axiom, length of instantiation chains, heuristic cost assignments to axioms). For many of our examples, root causes lie with individual events that throw the solver off. Identifying such events is hard, and to the best of our knowledge there currently is no debugging tool that helps do so.

We adopted a methodology, which we think can be largely automated, based on comparing solver traces using standard text diff tools. Given an unstable query, we could often find a "good" configuration (of solver parameters, variable names, etc.) that produces the expected behavior. When a good configuration does not appear organically, we had success with parameter fuzzing or randomness fuzzing through Mariposa. We then diffed the trace from a good run against one from a bad run, identified the first point of divergence, and made surgical modifications on the Z3 source to unify the solver's behavior over the two. Of course, the first point of divergence usually was not itself related to the root cause, but repeating this process until both configurations succeed allowed us to discover particularly elusive bugs.

This methodology would be greatly simplified if we were able to modify and *replay* solver traces. This would allow us to unify behavior without having to modify the solver's source code. Beyond trace diffs, replaying traces would make it much easier to test hypotheses of the form "had X been done differently, the query would not have exploded".

We also note that we used the tracing mechanism built into Z3, which was built for VCC, and not tailored for our purpose. There are many relevant steps that do not get logged (e.g. theory propagations, the state of the egraph, relevancy), which sometimes hides root causes and makes it hard to unify the next point of divergence.

5.3. Proof sketches

§4.1.1 uses a proof sketch to explain the root cause of instability for a simplified version of the `raw` query; we wrote similar sketches for most case studies. In our experience, writing down such pen-and-paper proof sketches and trying to identify what keeps the solver from finding them has been useful for debugging, as well as explaining root causes after the fact to convince oneself that the "right problem" was fixed. We believe debugging tools that ingest proof sketches along with solver traces and pinpoint the missing proof step would have greatly simplified our case studies.

Debugging instability through proof sketches mimics how one debugs functional correctness problems in most classes of software: by having a clear picture of the expected behavior and iteratively narrowing down where the program departs from it. Without proof sketches, this is only possible for SMT solvers at two extreme granularities, neither practical. On one extreme, one may say the expected behavior is

simply "the solver must return `unsat` quickly". This does not allow for iteratively narrowing down on the problem beyond the solver's interface. Debugging at the interface by giving the solver iteratively simplified/reduced versions of the query and trying to deduce which simplification step caused the problem often leads to ill-informed and misleading conclusions. On the other extreme, one may examine solver traces. This requires the developer to be intimately familiar with the solver's internals and even then they are often too large and too low-level to be practically understood by humans. Importantly, traces also mostly consist of "dead ends", i.e., branches explored and axioms instantiated by the solver as part of proof search that ultimately do not make progress towards the expected proof, and are uninteresting in pinpointing the divergence from expected behavior.

Proof sketches can be written at arbitrary granularity, and deepened only in steps that need to be narrowed down on (e.g., Fig. 1 elides the bitvector and arithmetic axioms that need to be involved for steps (8) and (10); the solver follows these steps as expected so the user does not need more detail). This makes proof-sketch debugging scale well with the size of the query being debugged. The original `raw` query spans 55× the number of SMT lines the simplified version in Fig. 1 does, but its proof sketch still consists of 10 (considerably bigger) steps.

6. Conclusion

We conjecture that the instability experienced by SMT-based verifiers today is mostly caused by fixable engineering problems, not fundamental theoretical limitations. We support this conjecture with 11 case studies, which identified three classes of root causes: solver bugs, misconfigurations, or misaligned expectations. We hope our experience will inform future work on addressing instability in three ways. First, some of the unstable queries we studied were not flagged as such by existing metrics; we argue for an instability metric that reflects user experience. Second, part of our debugging methodology can be automated, and would greatly benefit from tool support for debugging through proof sketches, and replaying solver traces. Third, having multiple sources of randomness makes instability debugging harder, so unifying all random behavior under explicit random seeds would be a step forward.

Acknowledgements

This material is based upon work supported by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL and DARPA.

Declaration on Generative AI

The authors did not use any generative AI tools in the writing of this paper.

References

[1] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, B. Parno, Mariposa: Measuring SMT instability in automated program verification, in: FMCAD, 2023.

[2] C. Hawblitzel, J. Howell, J. Lorch, A. Narayan, B. Parno, D. Zhang, B. Zill, Ironclad apps: End-to-end security via automated full-system verification, in: OSDI, 2014.

[3] A. Ferraiuolo, A. Baumann, C. Hawblitzel, B. Parno, Komodo: Using verification to disentangle secure-enclave hardware from software, in: SOSP, 2017.

[4] S. McLaughlin, G.-A. Jaloyan, T. Xiang, F. Rabe, Enhancing proof stability, in: Dafny Workshop, 2024.

[5] D. Amrollahi, M. Preiner, A. Niemetz, A. Reynolds, M. Charikar, C. Tinelli, C. Barrett, Using normalization to improve SMT solver stability, 2024.
[6] Y. Zhou, J. Bosamiya, J. Li, M. J. Heule, B. Parno, Context pruning for more robust SMT-based program verification, in: FMCAD, 2024.
[7] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, B. Parno, Verus: A practical foundation for systems verification, in: SOSP, 2024.
[8] Y. Ge, L. de Moura, Complete instantiation for quantified formulas in satisfiabiliby modulo theories, in: CAV, 2009.
[9] L. M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: TACAS, 2008.
[10] C. Cebeci, Y. Zou, D. Zhou, G. Candea, C. Pit-Claudel, Practical verification of system-software components written in standard c, in: SOSP, 2024.
[11] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, B. Parno, Storage systems are distributed systems (so verify them that way!), in: OSDI, 2020.
[12] L. de Moura, N. Bjørner, Relevancy Propagation, Technical Report MSR-TR-2007-140, 2007.
[13] Dafny Reference Manual: Measuring stability, https://dafny.org/v3.9.0/DafnyRef/DafnyRef.html#25751-measuring-stability, 2025.
[14] T. Bordis, K. R. M. Leino, Free facts: An alternative to inefficient axioms in Dafny, in: Formal Methods, 2024.
[15] K. R. M. Leino, C. Pit-Claudel, Trigger selection strategies to stabilize program verifiers, in: CAV, 2016.
[16] C. Cadar, D. Dunbar, D. R. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: OSDI, 2008.
[17] O. Padon, G. Losa, M. Sagiv, S. Shoham, Paxos made epr: decidable reasoning about distributed protocols, in: OOPSLA, 2017.
[18] C. Pulte, D. C. Makwana, T. Sewell, K. Memarian, P. Sewell, N. Krishnaswami, CN: Verifying systems C code with separation-logic refinement types, in: POPL, 2023.
[19] J. Hamza, N. Voirol, V. Kunčak, System FR: Formalized foundations for the Stainless verifier, in: OOPSLA, 2019.
[20] R. Willems, M. Schlaipfer, O. Bouissou, Helping users to reduce brittleness in their Dafny programs - a success story, in: Dafny Workshop, 2025.
[21] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: LPAR, 2010.
[22] K. L. McMillan, O. Padon, Ivy: A multi-modal verification tool for distributed algorithms, in: CAV, 2020.
[23] Y. Zhou, B. Parno, The Mariposa Benchmark, https://github.com/secure-foundations/mariposa-data, 2023.
[24] TPot query: raw.smt2, https://github.com/dslab-epfl/tpot/blob/unstable-queries/unstable-queries/raw.smt2, 2025.
[25] C. Cebeci, N. Bjørner, Can's fix to relevancy propagation, https://github.com/Z3Prover/z3/commit/8c5abdf818ebd674d57a8dc43f3fe04d73f6ab73, 2025.
[26] TPot query: april2nd.smt2, https://github.com/dslab-epfl/tpot/blob/unstable-queries/unstable-queries/april2nd.smt2, 2025.
[27] C. Cebeci, Mark fixed_eq literals as relevant, https://github.com/Z3Prover/z3/pull/7533, 2025.
[28] TPot query: offset.smt2, https://github.com/dslab-epfl/tpot/blob/unstable-queries/unstable-queries/offset.smt2, 2025.
[29] TPot query: bv_rewrite.smt2, https://github.com/dslab-epfl/tpot/blob/unstable-queries/unstable-queries/bv_rewrite.smt2, 2025.
[30] R. Leino, Z3 issue: Unexpected proof failure due to minor input tweak, https://github.com/Z3Prover/z3/issues/7444, 2024.
[31] K. McMillan, Z3 issue: Performance regression on stratified EPR formula, https://github.com/Z3Prover/z3/issues/7515, 2025.
[32] C. Cebeci, N. Bjørner, Fix complete_partial_func for finite domains, https://github.com/Z3Prover/

z3/pull/7533, 2025.
[33] D. Winterer, C. Zhang, Z. Su, Validating SMT solvers via semantic fusion, in: PLDI, 2020.
[34] B. Dutertre, L. de Moura, The Yices SMT solver, https://yices.csl.sri.com/papers/tool-paper.pdf, 2006.
[35] N. S. Bjørner, L. M. de Moura, Tractability and modern satisfiability modulo theories solvers, in: L. Bordeaux, Y. Hamadi, P. Kohli (Eds.), Tractability: Practical Approaches to Hard Problems, 2014.
[36] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. L. Roberts, S. Setty, B. Zill, IronFleet: Proving practical distributed systems correct, 2015.
[37] N. Amin, K. R. M. Leino, T. Rompf, Computing with an SMT solver, in: TAP, 2014.
[38] Jonáš Fiala, SMTScope, https://github.com/viperproject/smt-scope, 2024.
[39] N. Becker, P. Müller, A. J. Summers, The Axiom Profiler: Understanding and debugging SMT quantifier instantiations, in: TACAS, 2019.