Cazamariposas: Automated Instability Debugging in SMT-based Program Verification

Yi Zhou, Amar Shah, Zhengyao Lin, Marijn Heule, and Bryan Parno

Carnegie Mellon University, Pittsburgh, PA, USA
{yeet,amarshah,zhengyal,marijn,parno}@cmu.edu

Abstract. Program verification languages such as Dafny and F* often rely heavily on Satisfiability Modulo Theories (SMT) solvers for proof automation. However, SMT-based verification suffers from instability, where semantically irrelevant changes in the source program can cause spurious proof failures. While existing mitigation techniques emphasize preemptive measures, we propose a complementary approach that focuses on diagnosing and repairing specific instances of instability-induced failures. Our key technique is a novel differential analysis to pinpoint problematic quantified formulas in an unstable query. We implement this technique in Cazamariposas, a tool that automatically identifies such quantified formulas and suggests fixes. We evaluate Cazamariposas on multiple large-scale systems verification projects written in three different program verification languages. Our results demonstrate Cazamariposas' effectiveness as an instability debugger. In the majority of cases, Cazamariposas successfully isolates the issue to a single problematic quantifier, while providing a stabilizing fix.

Keywords: SMT · Program Verification · Proof Instability.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers play a prominent role in automated program verification. In verifiers such as Dafny [22], F* [32], or Verus [20], SMT solvers can often discharge complex verification conditions automatically, despite the generally undecidable program properties. In this way, SMT solvers significantly reduce the need for manual proof steps, facilitating the verification of large-scale systems [14,19,27].

However, the solvers must resort to incomplete heuristics to reason about undecidable program properties. Consequently, SMT-based program verifiers suffer from a persistent problem of *proof instability* [15,42], where trivial, non-semantic changes to the source program result in spurious proof failures. For instance, an SMT solver may fail to verify (i.e., return an inconclusive result on) a previously-proven verification condition after a simple variable renaming, even though the program's semantics remain unchanged.

Instability is a major impediment to the industrial deployment of automated verification [11]. In particular, instability disrupts the usual development work flow, incurring time and resource costs to debug spurious failures. Worse yet, the solver provides little insight as to why it rejects the modified query. As a result, the developer has little recourse beyond blindly modifying their source code in the hopes of nudging the solver into an accepting state [33].

While prior work [1,21,41] has tried to preemptively mitigate instability (with partial success—see Sec. 2), we propose a complementary approach to *repair* instability. In more detail, we automatically pinpoint the source of instability and suggest a query-specific *repair strategy*. Our approach is motivated by our finding that instability is often caused by only a few quantified formulas, among the thousands within a query; in fact, in 61% of the unstable queries we study, there is a *single* problematic formula to blame! By limiting the impact of just one ill-behaved quantified formula, we can repair most of the unstable proofs and avoid future failures.

We present a novel *differential analysis* to make such a diagnosis. We observe that, by definition [42], semantically equivalent variants of an unstable query lead to a mix of verification successes and failures. We thus compare the quantifier-instantiation profiles between succeeding and failing variants, focusing on quantifiers that are over-instantiated or under-instantiated in the failing cases. We further refine the analysis using novel proof and trace mining techniques, exploiting the causal relation between the instantiations.

We implement this approach in Cazamariposas, a debugging tool for SMT instability. Cazamariposas takes as input an unstable SMT query and outputs a concise report explaining the cause of the instability. Cazamariposas then provides a suggested repair strategy. For example, suppose Cazamariposas' analysis points to an over-instantiated quantifier ϕ within a query q. Cazamariposas would remove ϕ from q, confirm that the new query is now stabilized, and suggest that the developer remove the source-level construct that introduced ϕ to the query. In this way, Cazamariposas provides a fix that avoids further instability-related failures. Crucially, the repair strategy preserves the soundness of the verification condition (Sec. 4).

In Sec. 5, we evaluate Cazamariposas on a diverse set of benchmarks, with 615 unstable SMT queries collected from 12 system verification projects written in Dafny [22], F* [32], or Verus [20]. We find that Cazamariposas successfully repairs 70% of the unstable queries, of which 87% involve only a single quantifier.

2 Background and Related Work

SMT-based program verification has been gathering interest in academia [19,27] and industry [2,10]. Languages such as Dafny [22], F* [32], and Verus [20] enable mechanized proofs of program properties (e.g., functional correctness, memory safety) with the help of SMT solvers. In particular, developers write high-level proof hints for their programs, while the SMT solver can often automatically power through the rest of the proof.

In this section, we introduce the background on SMT-based program verification, define the notation for SMT-related concepts, and discuss prior work on proof instability.

2.1 Notation

We use a conjunctive formula $\Phi = \bigwedge_{i=0}^{n} \psi_i$ to represent an SMT query, where each ψ_i is an assertion. We slightly abuse the notation here by treating Φ as a set of assertions. For example, we use $\Phi \setminus \{\psi\}$ and $\Phi \cup \{\psi\}$ to denote a new query with an assertion ψ removed or added, respectively.

We assume the *goal-axioms* structure in program verification queries [41]. In particular, $\psi_0 = \neg\theta$ is the negation of the properties of the procedure under verification. Meanwhile, $\Lambda = \bigwedge_{i=1}^{n} \psi_i$ is a conjunction of *axioms* encoding the semantics of: **(1)** the verification language's constructs, and **(2)** other developer-written procedures that have already been verified. By checking that $\Phi = \Lambda \wedge \neg\theta$ is unsatisfiable, the SMT solver confirms that θ is a logical consequence of Λ.

The use of quantifiers is common in program verification. For ease of exposition, we use single-variable quantified formulas (e.g., $\phi = \forall x.\varphi$) as examples as long as it is clear how the method under discussion generalizes to quantification over multiple variables. We use Ω to represent the set of quantified formulas (including nested ones) in Φ.

For a universally quantified formula $\phi = \forall x.\varphi$, we use $\varphi[x \mapsto t]$ to denote the result of capture-free substitution of some ground term t for all free occurrences of x in φ. We refer to $\varphi[x \mapsto t]$ as an *instantiation* of ϕ, and t as the *instantiating term*. We use the calligraphic $\mathcal{I}^{\phi}$ to denote a set of instantiations of ϕ. For convenience, we define $\mathcal{I}^{\phi} = \{\}$ if ϕ is existentially quantified.

SMT-based verification languages rely heavily on pattern-based quantifier instantiation [24,25]. Each universally quantified formula $\forall x.\varphi$ is associated with (at least) one syntactic *pattern* π, where π would be a ground term expect for the free variable x in it. The body φ remains hidden until a ground term $\pi[x \mapsto t]$ enters the solver's context, at which point the solver creates the instantiation $\varphi[x \mapsto t]$. We refer to $\pi[x \mapsto t]$ as the *triggering term*.

2.2 Related Work

Proof instability is an obstacle to wide-scale industrial adoption of SMT-based verification. Galois highlights the "fragility of proofs" which can be "highly sensitive to minor changes in logical terms" [11]. Similarly, Amazon complains about the serious challenge of "lack of monotonicity and stability in runtimes" [29]. Numerous other large-scale verification projects cite SMT instability as a key pain point [2,8,10,12,15,19,28].

In prior work, researchers applied source-program-level analysis to choose better syntactic patterns for user-introduced quantified formulas, hoping to produce more stable SMT queries [21]. More recently, proposes using *free facts* [6], where quantified axioms are replaced by specific instantiations before the query is dispatched to the solver. Both techniques have shown some improvement, although they both rely on ad hoc measures of instability.

In our Mariposa [42] work, we presented a statistically rigorous approach to characterizing proof instability. At a high level, the Mariposa tool takes in an SMT query-solver pair (Φ, s), and outputs whether the query Φ is stable, unstable, or unsolvable when run with solver s. Mariposa generates *mutants* of the original query Φ, by (1) reordering assertions, (2) α-renaming variables, or (3) changing the random seed. Intuitively, if the solver's performance varies significantly across the mutants, Φ is unstable.

As part of the Mariposa project, we also curated a collection of program verification queries. In particular, we included several large-scale systems-verification projects written in Dafny [22] and F* [32] as a part of the Mariposa benchmark suite, where we found non-trivial amounts of instability.

In our follow-up work, we demonstrate that proof stability is strongly connected to the relevance of axioms [41]. In general, given a verification goal θ, if Λ is populated with unnecessary or irrelevant axioms, then the query is more likely to experience instability. We then introduce Shake, a context-pruning technique that reduces the number of irrelevant axioms in program verification queries, mitigating the instability on the Mariposa benchmark suite by 29% on Z3 and 41% on cvc5.

Amrollahi et al. preprocess an SMT query to put it into a canonical form [1] so as to normalize away semantically irrelevant source-level changes. They demonstrate mixed results, reducing instability on one Mariposa benchmark by 20%, while increasing it on another by 76%. Cumulatively, their approach increased instability by 4%.

In contrast to these approaches based on preemptive preprocessing, in this work, we propose to diagnose and repair specific instances of unstable proofs. Our approach draws inspiration from the field of automated theorem proving [17,30]. In particular, our query-specific diagnosis draws a parallel to the axiom selection problem [16], where the goal is to select a small subset of axioms from a large set of axioms to prove a theorem. Oftentimes, axiom selection is done using techniques from machine learning [18].

3 Motivating Examples

Programmers use metrics such as solver resource count and solving time as proxies for instability [33]. However, these heuristics are imperfect and often miss unstable proofs. Even when a programmer successfully detects instability, fixing it often requires considerable effort. When a proof in an automated verification language fails, conventional wisdom suggests various manual debugging techniques [26,33,34], including:

1. Adding source-level assertions to help guide the solver towards deriving important intermediate facts, and to trigger important quantifiers.

2. Adding source-level annotations to hide function definitions that are unnecessary for the proof. These annotations cause the verifier to encode the function such that the SMT solver treats it as uninterpreted.

We observe that these techniques target different problems at the SMT level. When the solver quickly returns `unknown` because of insufficient information, this may be addressed at the source level by adding source-level assertions. If the solver "times out" because it has spent too much time exploring extraneous parts of the proof space, this may be addressed at the source level by hiding unnecessary functions definitions.

In Sec. 4.3, we describe how Cazamariposas is able to automatically differentiate between the two cases above and suggest the appropriate fixes. In contrast, developers often struggle to find such fixes. To illustrate this, below we present two examples from the Verus benchmarks (Sec. 5.2) where Cazamariposas identifies a fix that the original programmers missed.

3.1 Diagnosing and Repairing Unnecessary Instantiations

Our first example, a lemma `lemma_from_after` written in Verus, has an unstable proof (see Appendix A.1 for the full example). In the original source code, the developer added a Verus annotation asking the solver to spin off a separate SMT query just for this lemma (normally Verus proves many goals incrementally in the same SMT context). Verus developers often use this annotation to try to improve stability, but Cazamariposas' measurements indicate it is still unstable.

To fix this proof's instability, the developer can try to hide various functions definitions using Verus's `hide` keyword. However, from the developer's perspective, it is not obvious which function might be to blame for the instability. The file containing `lemma_from_after` has 6 function definitions, and it imports 27 other Rust crates, each of which contributes many more functions that might be causing the instability. A priori, it is not even clear that hiding a function definition is the correct fix. The developer might instead guess that adding assertions to the body of `lemma_from_after` will improve stability.

In contrast, Cazamariposas explains that `make_stateful_set` is the cause, an unrelated function imported from a completely different crate. Cazamariposas suggests that the developer hide `make_stateful_set`, which produces a stable SMT query.

3.2 Diagnosing and Repairing Missing Instantiations

In the Verus code below, the function `entry_alive_wraps` on line 7 proves that every index `i` from `low` to `high` is alive if and only if `BUFFER_SIZE + i` is alive.[1] Liveness is defined via `entry_alive`, which among other operations, performs a division. At the SMT level, Verus represents division with the function `Euc_Div`, which is guarded to prevent division by 0. The original SMT query for `entry_alive_wraps` is unstable because the solver gets stuck going from `Euc_Div`

[1] We have simplified this example. The full example is in the Appendix A.2

to SMT-LIB's built-in division operator. As a developer, diagnosing such is quite difficult, let alone finding a fix.

```
1  const BUFFER_SIZE = ...
2
3  fn entry_alive(i: int) -> bool {
4   (i / BUFFER_SIZE) % 2 == 0
5  }
6
7  fn entry_alive_wraps(low: nat, high: nat)
8   ensures forall|i: nat|low <= i < high ==>
9     entry_alive(i, BUFFER_SIZE) == entry_alive(i + BUFFER_SIZE, BUFFER_SIZE)
10 {}
```

In contrast, Cazamariposas automatically diagnoses the problem at the SMT level and suggests a fix. At the SMT level, the ensures clause on `entry_alive_wraps` is negated, turning the universal quantifier on line 8 into an existential. Cazamariposas experiments with Skolemizing the quantified variable (i.e., turning it into a constant) and then searches for universal quantifiers in the query that might benefit from instantiation with the new Skolem constant. In this case, it identifies an instantiation of a quantifier about `Euc_Div` that stabilizes the proof. It suggests this fix to the developer, who can then use appropriate Verus-level syntax to apply the fix.

4 The Cazamariposas Methodology

Cazamariposas produces SMT-level *query edits* as repair strategies for proof instability. For example, given an unstable query Φ, Cazamariposas may pinpoint a quantified axiom ϕ_i such that $\Phi^* = \Phi \setminus \{\phi_i\}$ is stable. The removal of ϕ_i is sound because Φ^* maintains the original verification goal. The developer can therefore re-enact the SMT-level edits as source-level changes (e.g., hiding the procedure axiomtized by ϕ_i in this case), stabilizing the proof.

Cazamariposas follows an "edit-and-test" scheme to identify the repair strategies. Conceptually, for each quantified axiom $\phi_i \in (\Lambda \cap \Omega)$, Cazamariposas goes through the following:

- Hypothesize that quantifier reasoning over ϕ_i is the cause of instability.
- Select a *query edit* on Φ to reduce the reasoning obligations over ϕ_i.
- Apply the query edit to create a *candidate query* Φ^*.
- Test the stability of Φ^* using Mariposa.
- If Φ^* is not stable, dismiss the hypothesis for now.
- If Φ^* is stable, report ϕ_i as a cause of instability.

In Sec. 4.1, we discuss how we ensure that a query edit **(1)** weakens a particular target axiom, and **(2)** preserves the rest of the query context. Therefore, if Φ^* is stable, the edit is also a sound repair strategy.

While the "edit-and-test" scheme is conceptually simple, the vast number of quantified axioms makes it impractical to exhaustively test the stability impact of each ϕ_i individually. Therefore, in Sec. 4.2-Sec. 4.4, we describe how Cazamariposas effectively prioritize the likely suspects.

In Sec. 4.2, We make an observation on two distinctive failure modes in which instability manifests, corresponding to under and over-instantiated axioms. We then take advantage of the fact that an unstable query Φ has at least a passing mutant Φ_s and a failing mutant Φ_f. In Sec. 4.3, we propose metrics based on the instantiation profiles of Φ_s and Φ_f, highlighting quantified formulas that are excessively or insufficiently instantiated in Φ_f. In Sec. 4.4, we refine the analysis based on the failure mode with novel proof and trace mining techniques, exploiting the causal relation between the instantiations.

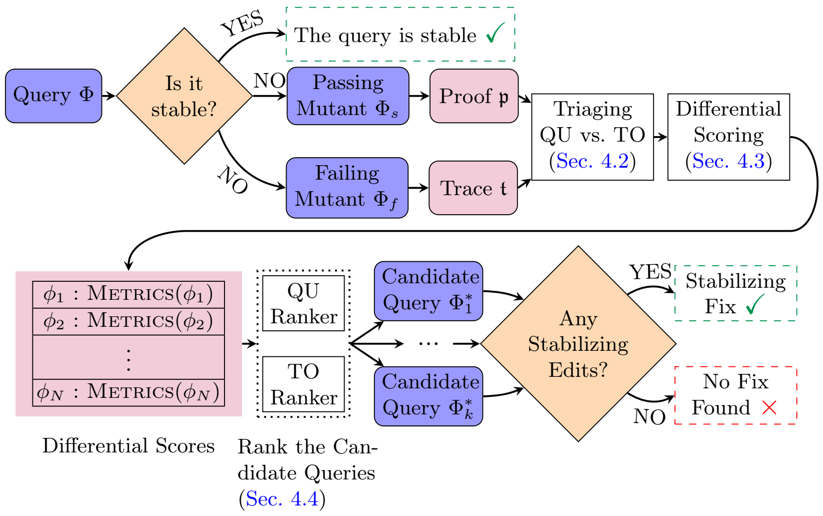


Fig. 1: The Design of Cazamariposas. We highlight SMT files in purple, other data in pink, Mariposa calls in orange, and Cazamariposas components in white. We write METRICS(ϕ_i) to represent the metrics in Sec. 4.3. For simplicity, we have omitted doubleton edits.

4.1 Testing the Axioms for Stability

We start with the methodology to evaluate the stability impact of individual axiom using query edits. More formally, we define a *singleton edit* as a pair (ϕ_i, a_i), where $\phi_i \in (\Lambda \cap \Omega)$, and a_i is an *action* among $\{\textbf{del}, \textbf{inst}, \textbf{inst-del}, \textbf{sk}\}$. Table 1 summarizes the actions we consider. In particular, for the actions **inst** and **ins-del**, we leverage the instantiation set $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ provided by the proof log; for the action **sk**, we use $f_{\phi_{ix}}$ to denote a Skolem constant from some existential assertion $\phi_i = \exists x.\varphi$.

Intuitively, the edits are meant to reduce or eliminate a solver's reasoning over ϕ_i, so a stabilized candidate also points to ϕ_i as a cause of instability. We now discuss some basic properties of the edit actions, including soundness, which ensures that a stabilizing edit is also a valid repair strategy.

Soundness. We define the soundness of the candidate Φ^* as:

$$\Phi^* \vdash \bot \implies \Phi \vdash \bot$$

Action a_i	Applicability ϕ_i	Edited Candidate Φ^*
del	$\phi_i \in (\Lambda \cap \Omega)$	$\Phi \setminus \{\phi_i\}$
inst	$\phi_i \in \Lambda, \phi_i = \forall x.\varphi$	$\Phi \cup \mathcal{I}_{\mathtt{p}}^{\phi_i}$
inst-del		$(\Phi \cup \mathcal{I}_{\mathtt{p}}^{\phi_i}) \setminus \{\phi_i\}$
sk	$\phi_i \in \Phi, \phi_i = \exists x.\varphi$	$(\Phi \cup \{\varphi[x \mapsto f_{\phi_i x}]\}) \setminus \{\phi_i\}$

Table 1: **Cazamariposas Query Edits**

We demonstrate soundness with a case analysis. When $a_i = \textbf{sk}$, the queries Φ^* and Φ are equivalent, so soundness trivially holds. Since we have restricted **sk** to be the only potential action on the goal, θ remains unchanged for the rest of the actions. Therefore, we could instead show that:

$$\Lambda^* \vdash \theta \implies \Lambda \vdash \theta$$

which holds as long as Λ^* is no stronger than Λ. When $a_i = \textbf{inst}$, because the elements of $\mathcal{I}_{\mathtt{p}}^{\phi_i}$ are tautological consequences of ϕ_i, Λ^* is as strong as Λ. When $a_i \in \{\textbf{del}, \textbf{inst-del}\}$, Λ^* might be weaker than Λ.

Completeness. We define completeness of the axiom set as follow:

$$\Lambda \vdash \theta \implies \Lambda^* \vdash \theta$$

The proof instantiation set $\mathcal{I}_{\mathtt{p}}^{\phi_i}$ is sufficient to establish θ by definition, so we maintain completeness. However, since the edits may weaken the axioms, we do sacrifice a broader sense of completeness. Specifically, **del** and **inst-del** may remove a quantified axiom ϕ_i, while the $\mathcal{I}_{\mathtt{p}}^{\phi_i}$ is only a finite subset of all possible instantiations of ϕ_i. Therefore do not guarantee that $\Lambda^* \vdash \phi_i$.

Composability. We not that if we perform a series of singleton edits $\Delta = \langle ..., (\phi_i, a_i), ... \rangle)$, we also maintain soundness and completeness. Intuitively, when instability arises from the interaction of multiple quantified axioms, singleton edits (i.e., $\|\Delta\| = 1$) might fall short to capture the cause, and thus we need to consider $\|\Delta\| \geq 2$. In the case where $\|\Delta\| = 2$, we call Δ a *doubleton edit*.

Practicality. In Cazamariposas, we focus on the SMT level to ensure applicability to multiple languages. Eventually, we would like to apply the edit actions to the source code, which we leave as future work. The query edits do generally correspond to source-level features in Dafny, F*, and Verus:

- **del** corresponds to source-level visibility control mechanisms. For example, Dafny's `opaque` keyword allows developers to hide the definition of a function by default.
- **inst** corresponds to quantifier instantiations as source-level annotations.
- **sk** corresponds to Hilbert's choice as a language construct. For example, Dafny's `var x :| P(x)` assigns `x` an arbitrary value such that `P(x)` holds.

However, the translation might not always be straightforward. As we discussed in Sec. 2, the axioms may also encode the semantics of language constructs. For example, **del** on an axiom for higher-order functions has no direct source-level equivalent. More specific to **inst**, if the repair adds a large number of instantiations to the source code, it is arguably impractical due to maintenance and readability concerns. Nevertheless, we have some empirical evidence that the repairs are often practical, which would make it interesting to explore automatic translation in the future.

Complexity. Another more pressing concern is the complexity of the search space. Consider a query Φ with n applicable singleton edits. The total number of potential candidate is roughly:

$$\sum_{i=1}^{\|\Delta\|} \binom{n}{i} = \binom{n}{1} + \binom{n}{2} + ... + \binom{n}{\|\Delta\|}$$

The combinatorial explosion makes it infeasible to test all candidates. Meanwhile, if a stabilizing edit involves too many axioms, it also become less realistic to reenact the repair at the source level.

Given the considerations, we limit our experiments to two classes: *singleton* and *doubleton* edits, i.e., $\|\Delta\| \leq 2$. Nevertheless, a massive search space remains for each class, with typically thousands of quantified axioms in a query. We thus further introduce a parameter k to limit the number of candidates we test for full stability. Specifically, we first test k singleton edits, and then k doubleton edits if the former fails to stabilize the query.

Figure 1 illustrates Cazamariposas's approach to efficiently navigate the search space of possible axioms and edits, addressing both challenges outlined above. We use our Mariposa tool [42] (Sec. 2) to produce semantically equivalent *mutants* of the original query. If the query is unstable, then some mutants will succeed, and some will fail. Hence we extract information about the quantifier instantiations produced in each group. We first use this information to triage (Sec. 4.2) which kind of instability-induced failure we face: either a *quick unknown* (QU) or a *slow timeout* (TO). In Sec. 4.3, we then leverage the solver's divergent behavior between a failed and a successful proof attempt to compute key metrics to identify the problematic quantifiers. Next, using the metrics and the failure mode, we select the query edits most likely to stabilize the query (**??**). Finally, we use Mariposa to evaluate the edits, and if any succeed in stabilizing the query, we report success and suggest the corresponding edit as a repair.

4.2 Triaging the Failure Mode

We begin our analysis by triaging the high-level cause of the underlying instability. Specifically, we observe two distinct modes of an instability-induced failure, which we name *quick unknown* (QU) and *slow timeout* (TO). In a QU, the solver quickly terminates (e.g., in < 1 second) with an `unknown` result, despite being given a generous timeout and resource limit. Meanwhile, for a TO, the solver runs on the query until it runs out of its time budget.

As we discuss in depth in Sec. 5.3, the two failure modes correspond to different instantiation profiles, signifying different underlying causes of instability. In a QU, the solver only explores a small number of instantiations before giving up. We therefore hypothesize that the solver is failing because it is missing key instantiations to prove the goal θ. In contrast, a TO is associated with orders of magnitude more instantiations, suggesting that the solver is spending significant time and resources on quantifier reasoning irrelevant to θ.

4.3 Calculating the Differential Metrics

We now introduce three metrics, namely Deficit, Excess, and Contingency, which measures the degree of insufficient or excessive instantiation. We define the metrics for quantified formulas in Ω, which is a superset of applicable assertions. In the next sections, we discuss how to aggregate metrics over Ω to rank potential edits over the quantified axioms.

Our analysis leverages a solver's divergent behavior between a failed and a successful verification attempt. By definition, if a query Φ is unstable, our Mariposa tool should find structurally isomorphic mutants, Φ_f and Φ_s, such that Φ_f fails and Φ_s succeeds. This allows us to compare the instantiation profiles between Φ_f and Φ_s modulo the isomorphism.

Concretely, we obtain from the solver a trace log $\mathfrak{t}$ for Φ_f, and a proof log $\mathfrak{p}$ for Φ_s. In theory, our method generalizes to multiple traces and proofs. In practice, collecting even one pair of $(\mathfrak{t}, \mathfrak{p})$ can entail difficulties. Hence, we focus our discussion on a pair of proof and trace. Furthermore, since we are comparing the instantiation profiles modulo the structural isomorphism between Φ_f and Φ_s, for the ease of exposition, we omit the detailed subscripts for $\mathfrak{t}$ and $\mathfrak{p}$, which would otherwise be $\mathfrak{t}_{\Phi_f}$ and $\mathfrak{p}_{\Phi_s}$.

We now define the metrics more formally. We use $\mathcal{I}^{\phi_i}_{\mathfrak{t}}$ and $\mathcal{I}^{\phi_i}_{\mathfrak{p}}$ to denote the instantiation set of ϕ_i in $\mathfrak{t}$ and $\mathfrak{p}$, respectively. For each quantified formula $\phi_i \in \Omega$, we compute the following:

- Deficit$(\phi_i, \mathfrak{p}, \mathfrak{t}) = \|\mathcal{I}^{\phi_i}_{\mathfrak{p}\text{-}\mathfrak{t}}\|$, where $\mathcal{I}^{\phi_i}_{\mathfrak{p}\text{-}\mathfrak{t}} = \mathcal{I}^{\phi_i}_{\mathfrak{p}} \setminus \mathcal{I}^{\phi_i}_{\mathfrak{t}}$. Intuitively, $\mathcal{I}^{\phi_i}_{\mathfrak{p}\text{-}\mathfrak{t}}$ is the set of instantiations in the proof but not in the trace. When $\|\mathcal{I}^{\phi_i}_{\mathfrak{p}\text{-}\mathfrak{t}}\|$ is large, the solver may be missing instantiations of ϕ_i that are crucial to reaching `unsat`.
- Excess$(\phi_i, \mathfrak{p}, \mathfrak{t}) = \|\mathcal{I}^{\phi_i}_{\mathfrak{t}\text{-}\mathfrak{p}}\|$, where $\mathcal{I}^{\phi_i}_{\mathfrak{t}\text{-}\mathfrak{p}} = \mathcal{I}^{\phi_i}_{\mathfrak{t}} \setminus \mathcal{I}^{\phi_i}_{\mathfrak{p}}$. Intuitively, $\mathcal{I}^{\phi_i}_{\mathfrak{t}\text{-}\mathfrak{p}}$ is the set of instantiations in the trace but not in the proof. When $\|\mathcal{I}^{\phi_i}_{\mathfrak{t}\text{-}\mathfrak{p}}\|$ is large, the solver may be wasting time and resources instantiating ϕ_i without making progress towards proving the goal.

We note that when IsExists(ϕ_i), the instantiation set $\mathcal{I}^{\phi_i}_{\mathfrak{t}}$ and $\mathcal{I}^{\phi_i}_{\mathfrak{p}}$ are both empty, so Deficit and Excess are trivially 0. In that case, we also introduce Contingency based on the instantiations that depend on ϕ_i:

- Contingency$(\phi_i, \mathfrak{p}) = \sum_{\phi_j \in \Omega} \|\{I \mid I \in \mathcal{I}^{\phi_j}_{\mathfrak{p}}, f_{\phi_{ix}} \sqsubseteq I\}\|$, where $f_{\phi_{ix}}$ is the Skolem constant of ϕ_i, $\phi_j \in \Omega$ is some (universally) quantified formula, and $I \in \mathcal{I}^{\phi_j}_{\mathfrak{p}}$ is some instantiation of ϕ_j containing $f_{\phi_{ix}}$. ($f_{\phi_{ix}} \sqsubseteq I$ denotes that $f_{\phi_{ix}}$ is a sub-term of I.) Intuitively, the metric reflects the proof instantiations that depend on $f_{\phi_{ix}}$. When ϕ_i has high Contingency, other quantified formulas cannot be sufficiently instantiated until ϕ_i is Skolemized.

Naively, we could already start prioritizing the quantified axioms based on these scores alone. Next we discuss how we aggregate the scores over the axioms, and choose the most promising edit actions for each axiom.

4.4 Ranking the Candidate Queries

As mentioned in Sec. 4.1, we use a parameter k to limit the number of candidates we test for full stability, where we first consider top-k singleton edits, and then doubleton edits if none of the singleton edits is stabilizing. We describe how to compute the scores for the singleton and doubleton edits in this section. The output of this stage are two partial maps, SScore and DScore.

1. We score each axiom ϕ_i, and then select an appropriate edit for it. When there are multiple possible actions on ϕ_i, we commit to one that is likely stabilizing. More formally, we create a partial map:

$$\textsc{SScore} = \{(\phi_i, a_i) \mapsto s_i \mid \phi_i \in \Phi\}$$

where $a_i \in \{\mathbf{del}, \mathbf{inst}, \mathbf{inst\text{-}del}, \mathbf{sk}\}$ is the chosen edit action.

2. We then score ordered pair of quantified assertions, along with the most promising actions for each assertion. More formally, we create another partial map:

$$\textsc{DScore} = \{\langle(\phi_i, a_i), (\phi_d, a_j)\rangle \mapsto s_{ij} \mid \phi_i, \phi_j \in \Phi\}$$

where $a_i, a_j \in \{\mathbf{del}, \mathbf{inst}, \mathbf{inst\text{-}del}, \mathbf{sk}\}$ are the chosen edit actions. $\langle(\phi_i, a_i), (\phi_j, a_j)\rangle$ is the doubleton edit we apply (in order). We note that both maps are partial because we may not find an applicable action for certain assertions.

We split the discussion on the ranking of edits based on the hypothesized failure mode (QU or TO), as the two failure modes require different strategies.

Ranking Edits for QU We start with how we handle QU failures, which is more straightforward. At the general triage (Sec. 4.2) stage, we hypothesize that the QU failures are due to the absence of certain instantiations. Intuitively, we are looking for under-instantiated axioms, where **inst** is applicable, i.e.,

$$\textsc{SScore} = \{(\phi_i, \mathbf{inst}) \mapsto s_i \mid \phi_i \in \Phi\}$$

There are various ways to use the differential scores to set s_i. Plausible contenders include:

1. $(\textsc{Deficit}, -\textsc{Excess})$
2. $(-\textsc{Excess}, \textsc{Deficit})$
3. $\kappa \cdot \textsc{Deficit} - \textsc{Excess}$ for some constant κ
4. $\textsc{Deficit}/\textsc{Excess}$

We experimented with multiple examples of each of these heuristics. Eventually we settled on the first one, using Deficit as the primary metric.

However, the picture becomes complicated when instantiations contain Skolem constants. In that case, we cannot fully materialize all of ϕ_i's proof instantiations $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$, unless all its Skolem dependencies are met. If the actual materializable instantiation count is 0 (i.e., that no instantiations can be created without Skolemization), then we drop ϕ_i in the singleton phase.

We address this issue in the doubleton stage. Specifically, we use the Contingency score to select the first axiom ϕ_i for **sk**; i.e., the quantified assertion with most "contingent" instantiations depending on it Skolem constant. When choosing the second axiom ϕ_j, we only consider ϕ_j candidates that depend on the Skolem constant $f_{\phi_{ix}}$ in their instantiations, and we can apply **inst** to ϕ_j. More formally,

$$\textsc{DScore} = \{\langle(\phi_i, \mathbf{sk}), (\phi_j, \mathbf{inst})\rangle \mapsto s_{ij} \mid \phi_i, \phi_j \in \Phi\}$$

where $s_{ij} = (\textsc{Contingency}(\phi_i, \mathfrak{p}), \textsc{Deficit}(\phi_j, \mathfrak{p}, \mathfrak{t}))$, and $\exists I \mid I \in \mathcal{I}_{\mathfrak{p}}^{\phi_j}, f_{\phi_{ix}} \sqsubseteq I$.

Ranking Edits for TO In the general triage stage (Sec. 4.2), we hypothesize that the solver is spending significant time and resources on irrelevant quantified formulas in a TO failure. For this failure mode, we focus on the *quantified axioms* in Λ as targets. Intuitively, we would need to suppress the excessive instantiation to stabilize the query, while editing the goal is not an option.

A natural choice would be to use the Excess for SScore, and then apply **del** to the axiom ϕ_i with the highest Excess. However, the situation is more complex than QU in two ways. (1) We cannot simply delete arbitrary ϕ_i with high Excess. The axiom may be necessary for the proof, and deleting it will render the goal un-provable (i.e., creating incompleteness). (2) Even if ϕ_i is indeed unnecessary, other excessively instantiated axioms may also be contributing to the instability.

Problem (1) is easier to address. We use the **inst-del** edit action, replacing the axiom ϕ_i with its instantiations from the successful proof trace $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$. Intuitively, this eliminates the need (and the ability) for the solver to instantiate ϕ_i: since $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ is sufficient for the proof, this action works around the incompleteness issue.

Problem (2) is more challenging. Anecdotally, if we focus solely on the Excess score, the debugging process turns into a "whack-a-mole" situation, where we delete one axiom, only to find another axiom with high Excess taking its place, and we fail to stabilize the query. Hence, to successfully repair the query, we need a mechanism to identify the underlying cause of the excessive instantiations.

Dependency Analysis In order to locate the root cause of TO instability. we further analyze the causal relations between the instantiations. Our notion of causality extends the *instantiation graph* from the SmtScope (formerly the Axiom Profiler) [4], a tool to analyze instantiation loops and other sources of poor performance in pattern-based SMT solvers. Below, we describe Axiom Profiler's approach and then our extension.

The instantiation graph is a directed acyclic graph over the terms (instantiations) in a trace log $\mathfrak{t}$. More formally, we model this graph G_0 with the node set:

$$\{(I, \phi_i) \mid I \in \mathcal{I}_{\mathfrak{t}}^{\phi_i}, \phi_i \in \Omega\}$$

where each instantiation is labelled with its quantified formula ϕ_i. Edges in the graph indicate the causal relations, which includes the following:

- **Instantiating Dependency**: an instantiation causes another one to materialize due to a matched pattern. Let (I_s, ϕ_s) and (I_d, ϕ_d) be two nodes in G_0, where $\phi_d = \forall x.\varphi_j$ is guarded by the pattern π_j. Suppose a sub-term of I_s matches π_j, i.e., $\pi_j[x \mapsto t] \sqsubseteq I_s$ for some ground term t. This match triggers the creation of $I_d = \varphi_j[x \mapsto t]$, corresponding to an edge $(I_s, \phi_s) \rightarrow (I_d, \phi_d)$ in G_1.
- **Equational Dependency**: an equational rewrite (from one instantiation) contributes to another instantiation. Continuing the example above, I_s may only trigger π_j after additional equality rewrites. Consider a quantified formula $\phi_{eq} = \forall x.p(x) \cong q(x)$ and one of its instantiations $I_{eq} = p(a) \cong q(a)$. The solver might have to rewrite I_s with I_{eq} first, where the rewrite result, $I_s[p(a) \mapsto q(a)]$, triggers the creation of I_d. In that case, there is also an edge $(I_{eq}, \phi_{eq}) \rightarrow (I_d, \phi_d)$ in G_0.

We further extend this graph G_0 from prior work into a graph G_1 to capture two additional types of dependencies.

- **Skolemizing Dependency**: a Skolem constant is a sub-term of an instantiation. Consider the existentially quantified $\phi_i = \exists x.\varphi_i$ with Skolem constant $f_{\phi_{ix}}$. There might be some node (I_d, ϕ_d) in G_1 such that $f_{\phi_{ix}} \sqsubseteq I_d$. In that case, we add the node $(f_{\phi_{ix}}, \phi_s)$, and the edge $(f_{\phi_{ix}}, \phi_s) \rightarrow (I_d, \phi_d)$ to G_1. This form of dependency follows the same intuition as in our definition of Contingency, except we apply it to the trace log here.
- **Nesting Dependency**: an instantiation is a (previously-nested) quantified formula, which creates further instantiations. For example, consider (I_s, ϕ_s), where $\phi_s = \forall x.(f(x) \wedge \forall y.g(x, y))$, and $I_s = f(t) \wedge \forall y.g(t, y)$ for some ground term t. Let $\phi_d = \forall y.g(t, y)$ be the nested quantified formula. Intuitively, I_s is the reason why ϕ_d exists at all. We thus add an edge from (I_s, ϕ_s) to every (I_d, ϕ_d), where $I_d \in \mathcal{I}_{\mathfrak{t}}^{\phi_d}$.

The graph G_1 captures the four types of dependencies we discussed above, which offers a rather low-level view of the instantiation reasoning in the trace. We further process G_1 so that it reflects the relation between the quantified formulas.

1. We collapse G_1 into a multi-edge graph G_2. We initialize G_2 with Ω as its nodes. For each edge $(I_s, \phi_s) \rightarrow (I_d, \phi_d)$ in G_1, we add an edge $\phi_s \rightarrow \phi_d$ to G_2.
2. We reduce G_2 into a weighted simple graph G_3. For each neighboring nodes ϕ_s and ϕ_d with $m_{s,d}$ parallel edges in G_2, we keep one edge $\phi_s \rightarrow \phi_d$ in G_3 with the weight $m_{s,d}$.
3. We normalize the edge weights in G_3, where we set the weight for $\phi_s \rightarrow \phi_d$ in G_3 to:

$$w_{s,d} = \frac{m_{s,d}}{\sum_{\phi_i \rightarrow \phi_d} m_{i,d}}$$

Intuitively, $w_{s,d}$ reflects the normalized "impact" of ϕ_s on ϕ_d over all the in-coming edges (via. other ϕ_i) to ϕ_d.

Hence the output of our dependency analysis is a directed simple graph G_3 over Ω, where each edge weight $w_{s,d}$ captures (or rather, approximates) the normalized impact of ϕ_s over ϕ_d. For example, $w_{s,d} = 0.5$ signifies that ϕ_s has an *immediate impact* on 50% of the instantiations of ϕ_d.

We then compute the transitive impact through fixed-point iterations. Concretely, for $\phi_i \in \Lambda$, we consider the reachable subgraph G_{ϕ_i} in G_3. We initialize a ratio $r_d = 0$ for each ϕ_d in G_{ϕ_i}, except for ϕ_i, where we set $r_i = 1$. We then update each ratio $r_d = \sum_s r_s \cdot w_{s,d}$. After the fixed-point computation terminates, we use the weighted sum of Excess as the final score for ϕ_i:

$$\textsc{SScore} = \{(\phi_i, a_i) \mapsto \sum_{\phi_j \in \Omega} \textsc{Excess}(\phi_j, \mathfrak{t}, \mathfrak{p}) \cdot r_j \mid \phi_i \in \Lambda\}$$

The fixed-point computation is non-decreasing by transitivity. However, there is no theoretical guarantee that it will converge. In particular, when G_{ϕ_i} contains a cycle, certain node's ratio may approach a limit at an exponential decay rate. Nevertheless, this is not a threat to practical usage. In particular, since floating point numbers represent the ratios, the convergence criteria must be threshold-based. For our implementation, we consider a ratio to have converged if its increment from the previous iteration is $\leq 10^{-4}$.

Now that we have the scores for each axiom, we proceed to choose the singleton edit action. We do so with a simple heuristic: if we can delete an axiom without causing incompleteness, we choose **del**. Otherwise, we instantiate the axiom with its proof instantiations, using **inst-del**. However, if there is Skolemization dependency preventing us from fully materializing the proof instantiations, we choose **inst** instead. Finally, if we have no other choice beyond Skolemization (**sk**), we do so.

Given the setup, ranking the doubleton edits is simple. We use the fixed-point computation to estimate the impact of each pair of axioms; i.e., we initialize $r_i = 1, r_j = 1$ for the pair (ϕ_i, ϕ_j), and then iterate over the nodes in G_3 to update the ratios. We then use the same weighted sum of Excess to calculate the final score for each pair. We also use the same heuristic to choose the edit actions for each pair.

5 Evaluation

In this section we perform an evaluation of Cazamariposas. We start with a brief overview of the implementation in Sec. 5.1, followed by a discussion of the benchmarks used in our evaluation Sec. 5.2. We then present the results of our evaluation, structured around two research questions: (1) Does the experimental data support our hypothesis about the different failure modes? and (2) How effective is Cazamariposas at identifying stabilizing edits?

5.1 Implementation

Our implementation of Cazamariposas is in 4,730 lines of Python, publicly available on Github [7], as a part of the Mariposa tool-chain. Here we also discuss our use of other tools and the configuration settings.

Mariposa We run Mariposa with its standard configuration, creating 180 total mutants for each query under full stability test. We use the default timeout of 60 seconds for the Mariposa benchmarks, but for the Verus benchmarks, we adjust the timeout limit to 10 seconds, the default for the Verus projects. We use Mariposa twice in our workflow, first to test whether the initial query Φ is stable and finally to test whether our various fixes are stable.

Z3 We use a recent version of the Z3 solver (4.13.0). We evaluate Cazamariposas with Z3, as Dafny, F*, and Verus are designed with Z3 in mind. Past work [42] has shown that other solvers such as cvc5 [3] and VAMPIRE [17] are not optimized for these verification languages, resulting in large numbers of unsolvable queries.

Z3 provides proof-production functionality, which can be complicated to use in practice. Enabling proof production often causes Z3 to take a different path, completely failing on an otherwise solvable query. To work around this, we have employed the following strategies: (1) use Z3 to find an unsatisfiable core first and then produce a proof from the core, (2) use 4 different versions of Z3 solver, (3) use an extended timeout of 1 hour, and (4) use up to 256 mutants. We note that these configuration are for finding proofs, not for testing stability. Despite all these, we were unable to get proofs for 4 Mariposa queries and 3 Verus queries. *In our evaluation, we count these cases as if* Cazamariposas *fails to find a fix.* Ideally the proof-production failures can be addressed within the SMT solver.

SMTSCOPE We created a fork of SMTSCOPE [13] to parse the Z3 trace logs, and then we enhance their instantiation graph as described in Sec. 4.4.

5.2 Verus Benchmark Set

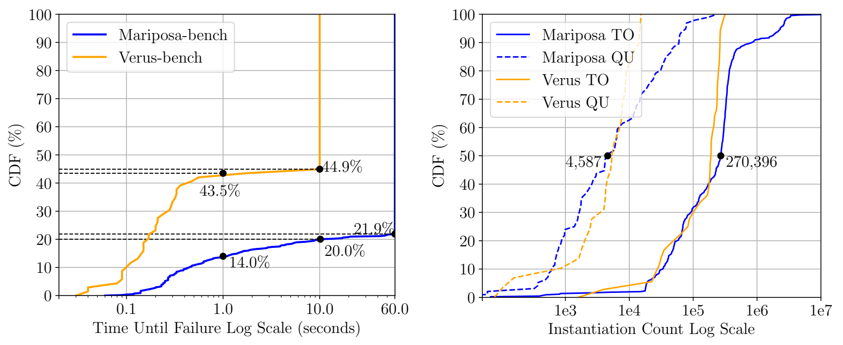
In addition to the Mariposa benchmark with 545 unstable queries (from five systems projects written in Dafny and F*), we curate a new benchmark set comprised of SMT queries from ten Verus verification projects (see Appendix B for details). Between these Verus projects, there are a total of 7,584 queries of which 7,514 (~99%) are stable and 70 (~1%) are unstable. We provide a detailed breakdown per project set in Appendix B.

We conduct all of our stability tests, for benchmark creation and evaluation, all on the same set of machines with an Intel Core i9-9900K (max 5.00 GHz) CPU, 128 GB of RAM, and Ubuntu 20.04.3.

In summary, we evaluate Cazamariposas on 70 unstable Verus queries and 545 unstable Mariposa queries, for a total of unstable 615 queries.

5.3 Failure Mode Distinction

Cazamariposas's first step is to triage a query's failure mode into either quick unknown (QU) or slow timeout (TO). Here we present empirical evidence for the



(a) Bimodal Mutant Failure Time (b) Instantiation Count Distributions

Fig. 2: **Failure Mode Distinction**. In (a) we plot the runtime of the failed mutants Φ_f, **before** the triage. In (b), we plot the instantiation count for the failed mutants Φ_f based on the failure modes triage.

distinction. In Figure 2a, for each original query Φ in the benchmarks, we report the runtime of its failed mutant Φ_f. The plot is in log scale, and we observe that the distribution for each benchmark is bimodal. For Verus-bench, ~43% of the failures occur within 1 second, barely any occur between 1 and 10 seconds, and the rest fail at 10 seconds. For Mariposa-bench, the distribution is more spread out, but the separation is still clear, where ~19% queries fail within 10 seconds, and ~78% time out after 60 seconds. There is almost no middle ground between the two modes. The x-axis of Figure 3 shows how many queries from each benchmark Cazamariposas ultimately classifies as QU versus TO.

In Figure 2b, we examine our hypothesis that QU failures are due to insufficient instantiation, and TO failures are due to excessive instantiation. We perform our triage, and then plot the instantiation counts based on the failure modes. Note the log-scale on x-axis, which highlights that the TO failures have orders of magnitude more instantiations than the QU failures. For example, in Mariposa TO, the median instantiation count is $270,396$ while in Mariposa QU, the median is $4,587$. The separation is also clear within Verus benchmark.

5.4 Stabilizing Edits Found

Next, we evaluate Cazamariposas' ability to automatically identify stabilizing edits. Recall that Cazamariposas first tries $k = 10$ singleton edits, and if none works, it tries 10 doubleton edits. Overall, Cazamariposas repairs 431/615 ($\approx$ 70%) of the benchmark queries. Figure 3 provides more details, reporting Cazamariposas' performance on the two benchmarks, subdivided by the underlying failure type (TO vs. QU). We note that Mariposa TO accounts for the largest absolute number of queries. Cazamariposas appears to be more effective on Verus queries in either failure type. Nevertheless, Cazamariposas repairs approximately

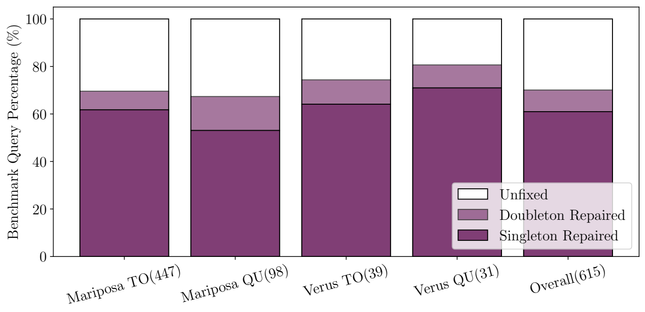
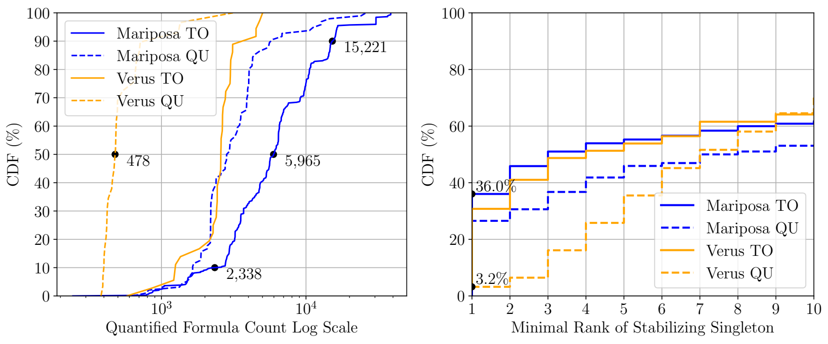


Fig. 3: Percentage of Benchmark Queries Repaired.

69% of the Mariposa queries and 77% of the Verus queries. This compares favorably to the best results from prior work (Sec. 2), which stabilized 29% of the Mariposa benchmark. We also observe that 375/615 ($\approx$ 61%) queries can be stabilized with a single edit. Doubleton edits subsequently provide a small but noticeable boost.

We also evaluate how well Cazamariposas ranks the stabilizing edits. First, in Figure 4a, we show the distribution of the number of quantified formulas in the original queries. For example, in Mariposa TO, the median count is 5, 965, which is a large search space for possible edits. The median count is lower in Verus QU, making it potentially more tractable to fully explore.



(a) Quantified Formula Count. (b) Rank of Stabilizing Edits.

Fig. 4: Finding Repairs Among Large Number of Quantified Formulas.

Now that we have sense of the search space, we evaluate how well Cazamariposas identifies the useful edits. In Figure 4b, we report the minimal rank of the stabilizing singleton edits. Specifically in singletons, given a query, Cazamariposas produces a ranked list of 10 edits, and we report the rank of the first stabilizing edit within this list. We note the endpoints of the CDFs on the y-axis. It is the probability that Cazamariposas finds a stabilizing edit within the first

10 singleton edits, which corresponds to Figure 3. We note the start points of the CDFs on the y-axis. This is the probability that the first edit Cazamariposas tries would directly work. For a Mariposa TO query, Cazamariposas has a 36% chance of finding a stabilizing edit with one shot.

6 Conclusions

Proof instability is a major impediment to industrial use of automated program verification tools. Hence, we propose a new approach that targets specific instances of instability. Using a novel differential analysis, we automatically classify the type of instability a query is experiencing, rank the problematic quantifiers in the query, and then efficiently identify targeted query edits that stabilize the query. We implement this approach in Cazamariposas and evaluate it on SMT queries from numerous verification projects written in three automated verification languages. Cazamariposas successfully repairs 70% of the unstable queries.

References

1. Amrollahi, D., Preiner, M., Niemetz, A., Reynolds, A., Charikar, M., Tinelli, C., Barrett, C.: Using normalization to improve SMT solver stability (2024), https://arxiv.org/abs/2410.22419
2. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Formal Methods in Computer Aided Design (FMCAD) (2018). https://doi.org/10.23919/FMCAD.2018.8602994
3. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Becker, N., Müller, P., Summers, A.J.: The axiom profiler: Understanding and debugging smt quantifier instantiations. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2019)
5. Bhardwaj, A., Kulkarni, C., Achermann, R., Calciu, I., Kashyap, S., Stutsman, R., Tai, A., Zellweger, G.: NrOS: Effective replication and sharing in an operating system. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 21) (Jul 2021)
6. Bordis, T., Leino, K.R.M.: Free facts: An alternative to inefficient axioms in Dafny. In: Formal Methods: 26th International Symposium (FM) (2024). https://doi.org/10.1007/978-3-031-71162-6_8
7. Cazamariposas. https://github.com/secure-foundations/mariposa, accessed Feb. 2025
8. Chakarov, A., Geldenhuys, J., Heck, M., Hicks, M., Huang, S., Jaloyan, G.A., Joshi, A., Leino, R., Mayer, M., McLaughlin, S., Mritunjai, A., Claudel, C.P., Porncharoenwase, S., Rabe, F., Rapoport, M., Reger, G., Roux, C., Rungta, N., Salkeld, R., Schlaipfer, M., Schoepe, D., Schwartzentruber, J., Tasiran, S., Tomb, A., Torlak, E., Tristan, J., Wagner, L., Whalen, M., Willems, R., Xiang, J., Byun,

T.J., Cohen, J., Wang, R., Jang, J., Rath, J., Syeda, H.T., Wagner, D., Yuan, Y.: Formally verified cloud-scale authorization. In: International Conference on Software Engineering (ICSE) (2025), https://www.amazon.science/publications/formally-verified-cloud-scale-authorization

9. Chen, X., Li, Z., Mesicek, L., Narayanan, V., Burtsev, A.: Atmosphere: Towards practical verified kernels in rust. In: Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification (KISV) (2023). https://doi.org/10.1145/3625275.3625401
10. Cutler, J.W., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A.M.: Cedar: A new language for expressive, fast, safe, and analyzable authorization. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (Apr 2024). https://doi.org/10.1145/3649835
11. Dodds, M.: Formally Verifying Industry Cryptography. IEEE Security and Privacy Magazine (2022). https://doi.org/10.1109/MSEC.2022.3153035
12. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2017). https://doi.org/10.1145/3132747.3132782
13. Fiala, J.: SmtScope (2025), https://github.com/viperproject/smt-scope, accessed Feb. 2025
14. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: Proving practical distributed systems correct. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2015)
15. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad Apps: End-to-end security via automated full-system verification. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (October 2014)
16. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: International Conference on Automated Deduction. pp. 299–314. Springer (2011)
17. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Computer Aided Verification (CAV) (2013)
18. Kühlwein, D., Blanchette, J.C.: A survey of axiom selection as a machine learning problem (2014), https://www.tcs.ifi.lmu.de/mitarbeiter/jasmin-blanchette/axiom_sel.pdf
19. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (November 2024)
20. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (December 2023)
21. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) Proceedings of the International Conference on Computer Aided Verification (CAV) (2016)
22. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (2010)

23. Lin, Z., Gancher, J., Parno, B.: Flowcert: Translation validation for asynchronous dataflow via dynamic fractional permissions. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (October 2024)
24. Moskal, M.: Programming with triggers. In: Proceedings of the Workshop on Satisfiability Modulo Theories (2009)
25. de Moura, L., Bjørner, N.: Efficient e-matching for SMT solvers. In: Conference on Automated Deduction (CADE) (2007)
26. Profiling Z3 and solving proof performance issues. https://fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html#profiling-z3-and-solving-proof-performance-issues
27. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., Kulatova, N., Ramananandro, T., Rastogi, A., Swamy, N., Wintersteiger, C., Zanella-Beguelin, S.: EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2020)
28. Reitz, A., Fromherz, A., Protzenko, J.: Starmalloc: Verifying a modern, hardened memory allocator. Proc. ACM Program. Lang. (Oct 2024). https://doi.org/10.1145/3689773
29. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Proceedings of the International Conference on Computer Aided Verification (CAV) (2022)
30. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)
31. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., Xu, T.: Anvil: Verifying liveness of cluster management controllers. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, Santa Clara, CA (Jul 2024), https://www.usenix.org/conference/osdi24/presentation/sun-xudong
32. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent Types and Multi-Monadic Effects in F*. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (2016)
33. Tomb, A., Tristan, J.B.: Avoiding verification brittleness in Dafny. https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/ (2023)
34. Verification debugging when verification fails. https://dafny.org/dafny/DafnyRef/DafnyRef#sec-brittle-verification
35. Verified IronKV. https://github.com/verus-lang/verified-ironkv, accessed Feb. 2025
36. Verified memory allocator. https://github.com/verus-lang/verified-memory-allocator, accessed Feb. 2025
37. Verified node replication. https://github.com/verus-lang/verified-node-replication, accessed Feb. 2025
38. Verified page table for NrOS. https://github.com/utaal/verified-nrkernel, accessed Feb. 2025
39. Verified splinter db. https://github.com/vmware-labs/verified-betrfs/tree/main/Splinter, accessed Feb. 2025
40. Verified storage. https://github.com/microsoft/verified-storage, accessed Feb. 2025

41. Zhou, Y., Bosamiya, J., Li, J., Heule, M., Parno, B.: Context pruning for more robust smt-based program verification. In: Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference (October 2024)
42. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) (October 2023)
43. Zhou, Z., Anjali, Chen, W., Gong, S., Hawblitzel, C., Cui, W.: VERISMO: a verified security module for confidential VMs. In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI) (2024)

A Full Code Examples

A.1 Unnecessary Instantiations Example

In the example in Sec. 3.1, the lemma `lemma_from_after` was shortened from `lemma_from_after_get_stateful_set_step_to_after_update_stateful_set_step`.

The full function definition is below. On line 33, we have the `hide` expression. This is not in the original program, but including it will stabilize the corresponding SMT query.

```
 1 #[verifier(spinoff_prover)]
 2 proof fn lemma_from_after_get_stateful_set_step_to_after_update_stateful_set_step(
 3   spec: TempPred<RMQCluster>, rabbitmq: RabbitmqClusterView, resp_msg: RMQMessage
 4 )
 5   requires
 6     spec.entails(always(lift_action(RMQCluster::next()))),
 7     spec.entails(tla_forall(|i| RMQCluster::controller_next().weak_fairness(i))),
 8     spec.entails(always(lift_state(RMQCluster::crash_disabled()))),
 9     spec.entails(always(lift_state(RMQCluster::busy_disabled()))),
10     spec.entails(always(lift_state(RMQCluster::pending_req_of_key_is_unique_with_unique_id
    (rabbitmq.object_ref())))),
11     spec.entails(always(lift_state(RMQCluster::every_in_flight_msg_has_unique_id()))),
12     spec.entails(always(lift_state(RMQCluster::each_object_in_etcd_is_well_formed()))),
13     spec.entails(always(lift_state(RMQCluster::desired_state_is(rabbitmq)))),
14     spec.entails(always(lift_state(helper_invariants::
    every_resource_update_request_implies_at_after_update_resource_step(SubResource::
    StatefulSet, rabbitmq)))),
15     spec.entails(always(lift_state(helper_invariants::
    stateful_set_not_exists_or_matches_or_no_more_status_update(rabbitmq)))),
16     spec.entails(always(lift_state(helper_invariants::
    no_delete_resource_request_msg_in_flight(SubResource::StatefulSet, rabbitmq)))),
17     spec.entails(always(lift_state(helper_invariants::
    cm_rv_is_the_same_as_etcd_server_cm_if_cm_updated(rabbitmq)))),
18     spec.entails(always(lift_state(helper_invariants::
    resource_object_only_has_owner_reference_pointing_to_current_cr(SubResource::
    StatefulSet, rabbitmq)))),
19     spec.entails(always(lift_state(helper_invariants::
    stateful_set_in_etcd_satisfies_unchangeable(rabbitmq)))),
20     spec.entails(always(lift_action(helper_invariants::cm_rv_stays_unchanged(rabbitmq)))),
21   ensures
22     spec.entails(
23       lift_state(|s: RMQCluster| {
24         &&& resp_msg_is_the_in_flight_ok_resp_at_after_get_resource_step(SubResource::
    StatefulSet, rabbitmq, resp_msg)(s)
25         &&& !sub_resource_state_matches(SubResource::StatefulSet, rabbitmq)(s)
26       })
```

```
      .leads_to(lift_state(|s: RMQCluster| {
        &&& pending_req_in_flight_at_after_update_resource_step(SubResource::StatefulSet,
    rabbitmq)(s)
        &&& !sub_resource_state_matches(SubResource::StatefulSet, rabbitmq)(s)
      }))
    ),
{
  hide(make_stateful_set);
  let pre = |s: RMQCluster| {
    &&& resp_msg_is_the_in_flight_ok_resp_at_after_get_resource_step(SubResource::
    StatefulSet, rabbitmq, resp_msg)(s)
    &&& !sub_resource_state_matches(SubResource::StatefulSet, rabbitmq)(s)
  };
  let post = |s: RMQCluster| {
    &&& pending_req_in_flight_at_after_update_resource_step(SubResource::StatefulSet,
    rabbitmq)(s)
    &&& !sub_resource_state_matches(SubResource::StatefulSet, rabbitmq)(s)
  };
  let input = (Some(resp_msg), Some(rabbitmq.object_ref()));
  let stronger_next = |s, s_prime: RMQCluster| {
    &&& RMQCluster::next()(s, s_prime)
    &&& RMQCluster::crash_disabled()(s)
    &&& RMQCluster::busy_disabled()(s)
    &&& RMQCluster::pending_req_of_key_is_unique_with_unique_id(rabbitmq.object_ref())(s)
    &&& RMQCluster::every_in_flight_msg_has_unique_id()(s)
    &&& RMQCluster::each_object_in_etcd_is_well_formed()(s)
    &&& RMQCluster::desired_state_is(rabbitmq)(s)
    &&& helper_invariants::
    every_resource_update_request_implies_at_after_update_resource_step(SubResource::
    StatefulSet, rabbitmq)(s)
    &&& helper_invariants::stateful_set_not_exists_or_matches_or_no_more_status_update(
    rabbitmq)(s)
    &&& helper_invariants::no_delete_resource_request_msg_in_flight(SubResource::
    StatefulSet, rabbitmq)(s)
    &&& helper_invariants::cm_rv_is_the_same_as_etcd_server_cm_if_cm_updated(rabbitmq)(s)
    &&& helper_invariants::resource_object_only_has_owner_reference_pointing_to_current_cr
    (SubResource::StatefulSet, rabbitmq)(s)
    &&& helper_invariants::stateful_set_in_etcd_satisfies_unchangeable(rabbitmq)(s)
    &&& helper_invariants::cm_rv_stays_unchanged(rabbitmq)(s, s_prime)
  };

  combine_spec_entails_always_n!(
    spec, lift_action(stronger_next),
    lift_action(RMQCluster::next()),
    lift_state(RMQCluster::crash_disabled()),
    lift_state(RMQCluster::busy_disabled()),
    lift_state(RMQCluster::pending_req_of_key_is_unique_with_unique_id(rabbitmq.object_ref
    ())),
    lift_state(RMQCluster::every_in_flight_msg_has_unique_id()),
    lift_state(RMQCluster::each_object_in_etcd_is_well_formed()),
    lift_state(RMQCluster::desired_state_is(rabbitmq)),
    lift_state(helper_invariants::
    every_resource_update_request_implies_at_after_update_resource_step(SubResource::
    StatefulSet, rabbitmq)),
    lift_state(helper_invariants::
    stateful_set_not_exists_or_matches_or_no_more_status_update(rabbitmq)),
    lift_state(helper_invariants::no_delete_resource_request_msg_in_flight(SubResource::
    StatefulSet, rabbitmq)),
    lift_state(helper_invariants::cm_rv_is_the_same_as_etcd_server_cm_if_cm_updated(
    rabbitmq)),
    lift_state(helper_invariants::
    resource_object_only_has_owner_reference_pointing_to_current_cr(SubResource::
    StatefulSet, rabbitmq)),
    lift_state(helper_invariants::stateful_set_in_etcd_satisfies_unchangeable(rabbitmq)),
    lift_action(helper_invariants::cm_rv_stays_unchanged(rabbitmq))
  );

  assert forall |s, s_prime: RMQCluster| pre(s) && #[trigger] stronger_next(s, s_prime)
    implies pre(s_prime) || post(s_prime) by {
    let step = choose |step| RMQCluster::next_step(s, s_prime, step);
    let resource_key = get_request(SubResource::StatefulSet, rabbitmq).key;
    match step {
      Step::ApiServerStep(input) => {
        let req = input.get_Some_0();
        assert(!resource_delete_request_msg(resource_key)(req));
        assert(!resource_update_request_msg(resource_key)(req));
        assert(!resource_update_status_request_msg(resource_key)(req));
      },
      _ => {}
    }
  }
  RMQCluster::lemma_pre_leads_to_post_by_controller(spec, input, stronger_next, RMQCluster
    ::continue_reconcile(), pre, post);
}
```

A.2 Missing Instantiations Example

In the example in Sec. 3.1, the function `log_entry_alive_wrap_around` was shortened to `entry_alive_wraps`, the function `log_entry_alive_value` was shortened to `entry_alive` and `buffer_size` was treated the same `LOG_SIZE`. Additionally, we omitted the function `log_entry_is_alive`.

Below we provide the full Verus code without any of those simplifications. Lines 46-56 are not present in the original file, but including them will stabilize the corresponding query.

This fix uses a `forall ... implies ...` block, which introduces a universally quantified variable `i` with a presupposition, and tries to prove an implication by allowing the programmer to make assertions about `i`. This is essentially a source-level version of Skolemizing.

The assertion on line 55 triggers the necessary instantiations for the `EucDiv` axiom.

```
1  pub open const LOG_SIZE: usize = 512 * 1024;
2
3  pub open spec fn log_entry_alive_value(logical: LogicalLogIdx, buffer_size: nat) -> bool
4    recommends
5      buffer_size == LOG_SIZE,
6  {
7    ((logical / buffer_size as int) % 2) == 0
8  }
9
10 /// predicate to check whether a log entry is alive
11 pub open spec fn log_entry_is_alive(
12   alive_bits: Map<LogIdx, bool>,
13   logical: LogicalLogIdx,
14   buffer_size: nat,
15 ) -> bool
16   recommends
17     buffer_size == LOG_SIZE,
18 {
19   let phys_id = log_entry_idx(logical, buffer_size);
20   alive_bits[phys_id as nat] == log_entry_alive_value(logical, buffer_size)
21 }
22
23 spec fn add_buffersize(i: int, buffer_size: nat) -> int {
24   i + buffer_size
25 }
26
27 proof fn log_entry_alive_wrap_around(
28   alive_bits: Map<LogIdx, bool>,
29   buffer_size: nat,
30   low: nat,
  high: nat,
)
  requires
    buffer_size == LOG_SIZE,
    forall|i: nat| i < buffer_size <==> alive_bits.contains_key(i),
    low <= high <= low + buffer_size,
  ensures
    forall|i: int|
      low <= i < high ==> log_entry_is_alive(alive_bits, i, buffer_size)
        == !#[trigger] log_entry_is_alive(
        alive_bits,
        add_buffersize(i, buffer_size),
        buffer_size,
      ),
{
  assert forall|i: int|
        low <= i < high
    implies
      log_entry_is_alive(alive_bits, i, buffer_size)
       == !#[trigger] log_entry_is_alive(
                alive_bits,
                add_buffersize(i, buffer_size),
                buffer_size)
    by {
      assert ((i + buffer_size)/buffer_size==(i + buffer_size)/buffer_size);
    }
}
```

B Verus Benchmark Projects

We curate a new benchmark set comprised of SMT queries from the following ten Verus verification projects.

- VerusKV is a distributed key-value store [35].
- VerusMimalloc is a concurrent memory allocator [36].
- VerusNR is a concurrent NUMA-aware node-replication library [37].
- VerusStorage is a storage system targeting persistent memory devices [40].
- VerusPT is an implementation [38] of the NrOS [5] page table.
- Atmosphere is a full-featured microkernel [9].
- VerusSplinterDB is a key-value store designed around a B^{ϵ} tree [39].
- Verismo is a security module for confidential VMs [43].
- Anvil is a tool for verifying Kubernetes controllers with Verus [31]. The project verifies three controllers: ZooKeeper, RabbitMQ, and FluentBit.
- FlowCert is a tool for the translation validation of dataflow programs [23].

Benchmark	**Stable**		**Unstable**		**Total**
	Number	*%*	*Number*	*%*	
VerusKV	363	100.00%	0	0.00%	363
VerusMimalloc	718	99.03%	7	0.97%	725
VerusNR	252	99.21%	2	0.79%	254
VerusStorage	374	94.44%	22	5.56%	396
VerusPT	336	99.41%	2	0.59%	338
Atmosphere	330	99.70%	1	0.30%	331
FlowCert	27	96.55%	1	3.45%	28
SplinterDB	1,213	99.84%	2	0.16%	1,215
Verismo	2,113	99.39%	13	0.61%	2,126
Anvil	1,788	98.89%	20	1.11%	1,808
Total	7,514	99.08%	70	0.92%	7,584

Table 2: Stability results on the Verus benchmark projects