Formal Methods in Computer-Aided Design 2024

Context Pruning for More Robust SMT-based Program Verification

Yi Zhou, Jay Bosamiya, Jessica Li, Marijn J. H. Heule, Bryan Parno
Carnegie Mellon University, Pittsburgh, PA, USA
{yeet,jaybosamiya,jgli,marijn,parno}@cmu.edu

***Abstract*—SMT solvers provide powerful proof automation for program verification. However, relying on SMT solvers also leads to proof instability, where a previously successful proof may fail after the developer makes trivial modifications to the source program. Such instability is a major headache for developers, but the causes and potential mitigations for it have received limited attention. In this study, we find that irrelevant query context accounts for 78% of the instability in existing program-verification query sets. As a result, we design SHAKE, a novel technique that leverages the structure in program-verification SMT queries in order to filter out irrelevant context from such queries. SHAKE is the first SMT-level technique that targets instability, and we implement it as a pre-processing step for SMT solvers. We evaluate SHAKE on real-world, large-scale query sets, and we find that it leads to large reduction in context and a 29% and 41% improvement in query stability on Z3 and cvc5, with minor performance overhead.**

I. Introduction

Satisfiability Modulo Theories (SMT) solvers play a crucial role in *automated program verification*, since verification-oriented languages (e.g., Dafny [1] or F* [2]) often translate program source code and specifications into verification conditions [3], [4] that they encode as SMT queries [5]. Essentially, each SMT query states that the code adheres to its specifications, and the SMT solver (e.g., Z3 [6] or cvc5 [7]) checks if this statement holds. The solvers obviate many manual proof steps, simplifying the verification of large code bases [8]–[14].

Unfortunately, SMT-based program verification is not necessarily robust. Notably, the approach is susceptible to *proof instability* [15], where trivial changes to the program cause spurious verification failures. For instance, the SMT solver may reject a previously-verified program after the developer renames a variable, even though the program's semantics clearly did not change. Faced with such a proof failure, the developer may need to tediously provide manual proof steps to guide the solver back on track [16], which arguably defeats the purpose of automation. To the frustration of practitioners, instability has been a long-standing problem [15], [17]–[23].

While instability is pervasive in practice [15], its causes remain understudied, let alone its mitigation. Existing literature has pointed at several potential culprits [18]–[20], but these claims are anecdotal and lack quantitative evidence.

In this work, we explore the problem quantitatively and find that irrelevant query context is a *major contributor* to instability. Our experiments on unsatisfiable cores from a large-scale program-verification query set discover that typically 96%–99% of the assertions in a query do not remain in the unsatisfiable core—they are irrelevant to verification. More importantly, irrelevant assertions account for 78% of the observed unstable instances (§III).

Motivated by the findings, we propose a novel SMT context-pruning technique, named SHAKE, to improve stability. We base SHAKE on the insight that program-verification tasks are typically automated theorem proving (ATP) [24] tasks, meaning that the verification queries are each composed of a goal assertion along with axiom assertions. SHAKE triages the axioms with respect to the goal and prunes the less relevant axioms.

While SMT solvers are built for constraint-solving, adopting a theorem-proving perspective helps improve stability. We implement SHAKE as a preprocessor, and evaluate it on large-scale program-verification query sets from the Mariposa study [15]. We find that SHAKE typically reduces the context by 3–10×. Moreover, we show that SHAKE can mitigate instability on Z3 by 29% and on cvc5 by 41%. SHAKE imposes little runtime overhead, even improving the number of solved instances on cvc5 by 73% in one benchmark and 8% overall.

In summary, we make the following contributions.

- We empirically show that irrelevant context is a major source of instability in program-verification queries.
- We propose a novel pruning technique, SHAKE, based on a theorem-proving view of program verification.
- We show that SHAKE reduces instability by 29%–41% on existing query sets, with only minor overhead.

To facilitate research on context pruning and instability mitigation, our source code and query sets are all available at https://github.com/secure-foundations/mariposa.

II. Background

Formal verification provides strong guarantees about program properties such as security and functional correctness. In recent years, academia has made notable progress in verifying large-scale systems [8]–[14], [25]–[28]. Industry has also adopted verification in certain mission-critical scenarios [29]–[31]. In particular, automated verification languages have gained popularity, exemplified by Dafny and F*, which are maintained by Amazon Web Services and Microsoft Research respectively.

These automated verification languages are powered by SMT solvers. Typically, a language's verification condition generator (VCG) encodes the source program into a logical formula, which states that the program's specification holds; i.e., the program is correct. If the SMT solver reports the negation of

https://doi.org/

the formula to be *unsatisfiable*, the program's specification is never violated, and thus the program verifies. However, since program properties are generally undecidable, the solver cannot guarantee that it will verify every correct program.

This incompleteness then leads to the phenomenon of *proof instability*, where a previously successful verification spuriously fails after trivial modifications to the source program. This happens because source-level changes obligate the VCG to create a new query for the SMT solver. Due to incompleteness, the solver may succeed on an old version of the query but may fail on the new one, even if the queries are semantically equivalent.

Instability is a major headache for developers. For individuals, it disrupts their incremental development process by diverting them from their main development tasks. For teams, instability is even more problematic, as instability may only appear when concurrent changes to the source code are merged.

In light of this problem, the Mariposa project [15] aims to quantify instability in SMT-based program verification. For a given SMT query-solver pair (q, s), the Mariposa tool outputs a stability category: `stable`, `unstable`, or `unsolvable`. In some cases the status may be `inconclusive`, which indicates that Mariposa does not have sufficient statistical power to confidently assign a category.

The Mariposa tool derives the stability status from the performance of s on q's mutants, which are semantically equivalent to q. Specifically, Mariposa creates the mutants by shuffling the assertions or renaming the symbols in q, as well as by reseeding the random number generator in s.

The Mariposa project experimented with large-scale program verification query sets. For this study, we use the Mariposa methodology to measure instability, and we also conduct our experiments on the Mariposa query sets. We exclude one query set, Komodo$_S$, from our study, which we discuss in §VII.

III. Query Context

In this section, we study the connection between query context and stability. We abstract an SMT query's context as a set of assertions, each introducing a constraint to the query. We then analyze each query's unsatisfiable core. Upon reaching an unsat result, the solver produces a core, which is a subset of the original assertions that the solver used to derive the unsat result. Thus, the solver-produced core serves as an oracle of relevant assertions, and what is excluded from the core can be considered irrelevant.

In §III-A, we describe our method to obtain unsat cores. In §III-B, we show that often only a tiny fraction of the assertions are relevant to verification success. In §III-C, we show that irrelevant context can be a major source of proof instability. In §III-D, we present a simple theorem-proving view of the query context and discuss how that view can help cut down on irrelevant assertions to improve stability.

A. Export the Unsatisfiable Core

In theory, we can export an unsat core by enabling an SMT solver's `produce-unsat-cores` option. In reality, obtaining an unsat core can sometimes be non-trivial, especially on unstable queries. Though uncommon, two types of problems may occur, so we document our workarounds here.

Unsuccessful Export. The solver might not be able to produce a core. There can be several reasons. First, the solver behaves differently depending on whether the core is requested or not. We have observed cases in which the solver returns `unsat` on a query, but returns `unknown` when core production is enabled. Second, the query itself might be unstable, meaning that the original query may fail, but some mutants of it may succeed. Third, a query might be completely unsolvable (regardless of mutations) with a particular solver version, but solvable with another.

In these cases, we perform Mariposa-style mutations to the query, attempting to obtain a core from any of the mutants. We then map the core from a successful mutant back to a core of the original query. If necessary, we also try the core export using different versions of the solver.

Incomplete Core. The solver might also produce a core query that is incomplete. Specifically, the solver might return `unsat` on the original query and successfully produce a core query; however, when given the core query, the solver fails to produce `unsat`, even with mutations applied to the core. This could be due to certain assertions that are necessary to the proof but missing in the core. Note that incompleteness here is not a strictly formal notion, since we do not have a ground truth for necessity.

When this happens, we apply a best-effort search to repair the core by adding assertions back to the core query, performing a bisection search to find a small addition of assertions that make the solver return `unsat` on the core. In practice, we find the incompleteness problem to occur more often with F* queries ($\sim 8\%$), and the core is typically only "missing" a small number (≤ 5) of assertions.

In summary, if the two issues above occur, we make a best-effort attempt to find a core query such that: its assertions form a subset of the original's, and it is sufficient for the solver to show `unsat`. We are successful in these attempts for all but a small fraction of the original queries. In that remaining fraction, we use the original query as the core query.

B. Most of the Context is Irrelevant

After acquiring an unsat core, we compare its context to the original. As shown in Figure 1, the original query context typically contains thousands of assertions. Using the assertion count as a proxy for the "size" of the context, we examine the *relevance ratio*:

$$\frac{\text{\# core assertions}}{\text{\# original assertions}} \times 100\%$$

Since an unsat core is a subset of the original query, the lower this ratio is, the less context is retained, and the more irrelevant context the original query has.

Figure 2 shows the CDFs of the relevance ratios for different projects. For example, on the left side lies the line for DICE$^\star_F$. The median relevance ratio (MRR) is 0.06%, meaning that

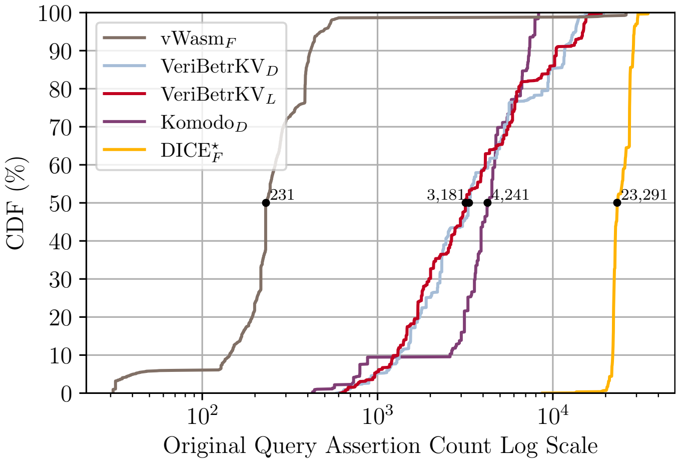

Fig. 1. **Original Query Assertion Count.** More to the right means larger query contexts, which may each contain thousands of assertions.

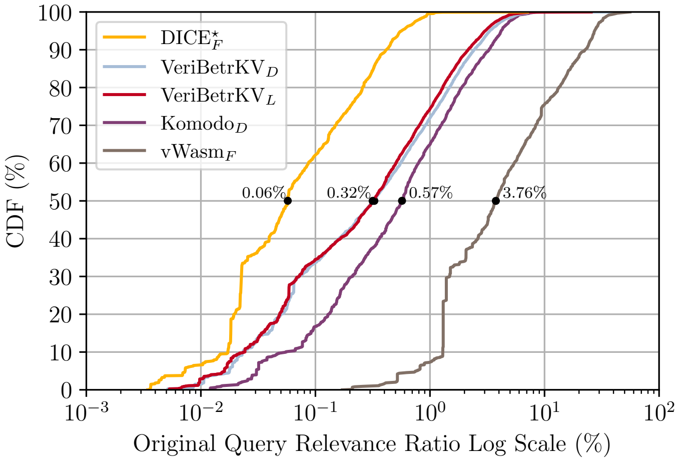

Fig. 2. **Original Query Context Relevance.** More to the left means more irrelevant contexts. Typically, the vast majority of an original query context is irrelevant.

for a typical query in the project, only 0.06% of the context is relevant. In vWasm$_F$, the MRR is 3.76%, which is almost an of order of magnitude higher than the other projects. We attribute this to the manual context tuning by the authors of vWasm$_F$, who explicitly documented the tedious effort [14], [15]. Nevertheless, if we consider the complement of the relevance ratio, typically 96.23–99.94% of the context is irrelevant, even considering vWasm$_F$.

C. Irrelevant Context Harms Stability

Given the significant amount of irrelevant context, we further analyze how that impacts stability. Here we compare and contrast the stability of the original queries and their cores. Recall the Mariposa stability status for a query-solver pair can be one of `unsolvable`, `unstable`, `stable`, or `inconclusive`. Given an original query q and its core q_c, we introduce the following two metrics:

- **Preservation**: given that q is `stable`, the probability that q_c remains `stable`.
- **Mitigation**: given that q is `unstable`, the probability that q_c becomes `stable`.

We use the Mariposa tool [32] with Z3 version 4.12.5 in this experiment. In Figure 3, we list the number of original queries and the scores for solver-produced core. As an example, in the original Komodo$_D$ queries, 1,914 are stable and 93 are unstable. In its core counterpart, 99.4% of the stable queries remain stable, while 90.3% of the unstable ones become stable. vWasm$_F$ is the only case where the core has no mitigation effect. However, its original queries are rarely unstable. As we noted previously, vWasm$_F$ also starts with more relevant original context. Therefore, the stability of vWasm$_F$ can be explained by the manual tuning done by the original developers.

Project	Original		Solver-Produced Core	
	Stable	Unstable	Preservation	Mitigation
Komodo$_D$	1,914	93	99.4%	90.3%
VeriBetrKV$_D$	4,983	172	99.5%	64.5%
VeriBetrKV$_L$	4,999	256	99.6%	83.6%
DICE$^\star_F$	1,483	20	99.6%	90.0%
vWasm$_F$	1,731	4	99.7%	0.0%
Overall	15,110	545	99.5%	78.3%

Fig. 3. **Stability of Core Queries.** Typically an unsat core preserves the stability of the original query, and it mitigates instability in 78.3% of the unstable queries.

Generally, the solver-produced unsat core is highly likely to preserve query stability. Moreover, across all projects, 78.3% of the unstable instances can be mitigated by using the core. In other words, irrelevant context is **a major contributor to instability**. This result suggests a promising mitigation strategy of pruning irrelevant assertions. In the next section, we discuss the composition of query context and how it can inform context pruning.

D. Context Pruning is Axiom Selection

In §II, we offered an overview of the verification condition generator (VCG) in automated verification languages. Here we give a more formal treatment on how a VCG constructs the query context, along with an intuitive view of the query as a theorem-proving task and context pruning as axiom selection.

The VCG typically creates an SMT query per procedure[1] under verification. Given a procedure P, the VCG encodes a verification goal ψ, which is a formula stating that P is correct. $\neg\psi$ is then placed into the query as an assertion. In practice, the goal ψ is rarely self-contained, since P usually refers to other procedures or relies on language-level axioms. The VCG also includes these dependencies in the query. As a result, the query context is a constraint set $\Gamma = \{\neg\psi\} \cup \Gamma_A$, where $\Gamma_A = \{\varphi_1, ..., \varphi_n\}$ is a set of axioms.

The standard semantics of an SMT query is the satisfiability of the constraint set Γ. We can interpret the query as checking the validity of $\Gamma \vdash \texttt{false}$, which is equivalent to $\Gamma_A \vdash \psi$.

[1] We generically refer to a function-like construct with pre/post-conditions as a procedure. It can be a function, method, lemma, etc.

Intuitively, this is a theorem-proving task, where the axioms in Γ_A are given to prove the verification goal ψ.

Through this view, the context pruning problem becomes an axiom selection problem, in which we choose a subset of axioms $\Gamma_R \subseteq \Gamma_A$ s.t. $\Gamma_R \vdash \psi$. The SMT solver usually takes the constraint-solving view of the query, where the relevance of an assertion is determined by its contribution to the unsatisfiability. As it turns out, the solver can also benefit from this theorem-proving perspective, where we define the relevance of the axiom assertions with respect to the goal.

IV. SHAKE

In this section, we introduce SHAKE, a pruning technique for SMT queries produced during program verification. At a high level, SHAKE takes a query as input and computes the distance from each axiom to the goal, indicating the relevance. More formally, the input to SHAKE is a set of constraints $\Gamma = \{\varphi_0, ..., \varphi_n\}$. For convenience, let $\varphi_0 = \neg\psi$, where ψ is the verification goal, while $\varphi_1, ..., \varphi_n$ are the axioms. The output of SHAKE is thus a map of distances:

$$dists = \{(\varphi_0 : 0), ..., (\varphi_n : d_n)\}$$

where the goal is at 0. SHAKE then prunes the axioms based on their distances. We first introduce a naive version of SHAKE, then progressively improve upon the design.

A. *The Naive* SHAKE *Algorithm*

In this version of SHAKE, we abstract a formula ϕ via the set of query-defined symbols it contains, denoted as $\textsc{Symbols}(\phi)$. More precisely, the symbols are the functions, constants, and datatypes introduced by the query, excluding sorts, local variables, and built-in SMT-LIB functions: intuitively, ubiquitous functions like $<$ or `not` do not convey much information.

Alg. 1 shows the naive SHAKE algorithm. We first initialize a context symbol set S_{ctx} from the goal. We then select all axioms φ_i such that $\textsc{Symbols}(\varphi_i)$ intersects with S_{ctx}, on the theory that intersection conveys relevance. After scanning through all axioms in this round, we augment S_{ctx} with the symbols from the selected axioms. The update is delayed until the end of the round, so S_{ctx} remains the same during this scan. Otherwise, the scan order would affect the content of S_{ctx}, introducing a form of instability.

Applying this process repeatedly scores the distance of an axiom φ_i based on the round in which $\textsc{Symbols}(\varphi_i)$ first intersects with S_{ctx}. The outer iteration continues until we reach a fixed point. When there are unreachable axioms at the end, they are assigned a distance of round count plus one.

In practice, we find that naive SHAKE typically terminates after very few iterations, giving little differentiation between axioms. The problem arises because naive SHAKE is too eager in its expansion. Since we use symbol sets to abstract away formulas, a single complex axiom with a large symbol set can easily saturate S_{ctx}, ending the process quickly. In light of this problem, we refine the formula abstraction to handle quantifiers, which SHAKE expands lazily.

Algorithm 1 Naive SHAKE

```
procedure NaiveShake($\Gamma = \{\varphi_0, ..., \varphi_n\}$)
    # assuming $\varphi_0$ is the goal
    $S_{ctx} \leftarrow \textsc{Symbols}(\varphi_0)$
    $dists, round \leftarrow \{(\varphi_0 : 0)\}, 1$
    repeat
        $acc \leftarrow \emptyset$
        for $\varphi_i \in \Gamma$ do
            if $S_{ctx} \cap \textsc{Symbols}(\varphi_i) \neq \emptyset$ then
                # check if $\varphi_i$ has been assigned a distance
                if $\varphi_i \in \textsc{Unreached}(dists, \Gamma)$ then
                    $dists \leftarrow dists \cup \{(\varphi_i : round)\}$
                $acc \leftarrow acc \cup \textsc{Symbols}(\varphi_i)$
        # update the symbol set after considering all $\varphi_i$
        $S_{ctx} \leftarrow acc \cup S_{ctx}$
        $round \leftarrow round + 1$
    until $\textsc{IsFixedPoint}(dists)$
    $max\_dist \leftarrow round + 1$
    for $\varphi_i \in \textsc{Unreached}(dists, \Gamma)$ do
        # assign maximum distance to unreachable axioms
        $dists \leftarrow dists \cup \{(\varphi_i : max\_dist)\}$
    return $dists$
```

```
(declare-fun foo (Int) Int)
(declare-fun bar (Int) Int)
(declare-fun qux (Int) Int)
(assert (forall ((x Int))
            (! (< (foo x) (bar (qux x)))
              :pattern ((foo x))
              :pattern ((bar x)))))
```

Fig. 4. **Example SMT Assertion with Pattern**. The patterns are hints to the solver on when to instantiate the quantifier. In this example, either the pattern `(foo x)` or the pattern `(bar x)` should be matched.

B. SHAKE *with Quantifiers*

In the queries we study, quantifiers often come with *patterns* [33], [34]. Patterns are syntactic hints to the solver as to when a quantifier should be instantiated; if the patterns are not matched, the quantified body remains hidden. In this version of SHAKE, we use the available patterns to refine the notion of relevance for formulas.

In this version, we construct a **formula state** for a given formula ϕ. We denote this via $\textsc{InitFState}(\phi)$, which augments ϕ with two fields:

- $\phi.S_{visible}$: the set of symbols in ϕ not under any quantifier.
- $\phi.qstates$: a list of **quantifier states**, constructed only from the outermost quantifiers in ϕ. The construction via $\textsc{InitQState}$ is lazy, meaning that any nested quantifiers are hidden under the outermost quantifier states.

Given a quantified formula ω, $\textsc{InitQState}(\omega)$ creates a quantifier state containing ω and two additional fields:

- $\omega.patterns$: a list of symbol sets from the patterns.
- $\omega.\phi_{hidden}$: the quantified body, which remains uninitialized until expanded, including any nested quantifiers it may contain.

For example, in Figure 4, the list of pattern symbol sets is $[\{$`bar`$\}, \{$`foo`$\}]$, and the hidden body is the formula `(< (foo x) (bar (qux x)))`.

SHAKE is lazy when determining the relevance of a quantifier state, reflected in the TRYEXPAND procedure. Given a symbol set S, if none of the $\omega.patterns$ is a subset of S, the quantifier is irrelevant, and ϕ_{hidden} remains unexpanded (i.e., SHAKE ignores the symbols it contains). The subset condition is necessary because for an actual instantiation, all the symbols in a specific pattern must be present in S. Upon a match, TRYEXPAND creates a new formula state from its hidden body ϕ_{hidden}. We note that the quantifier is only expanded by one level of nesting via INITFSTATE.

```
procedure TRYEXPAND($\omega, S_{ctx}$)
    $relevant \leftarrow$ false
    # subset check needed to check for pattern match
    for $S \in \omega.patterns$ do
        if $S \subseteq S_{ctx}$ then
            $relevant \leftarrow$ true
    if $relevant$ then
        # create a new formula state from the hidden body
        INITFSTATE($\omega.\phi_{hidden}$)
        return SOME($\omega.\phi_{hidden}$)
    return NONE
```

SHAKE checks the relevance of a formula state φ as follows. Given a symbol set S, φ is relevant if $\varphi.S_{visible}$ intersects with S, or if any of the $\varphi.qstates$ is considered relevant. When SHAKE expands a quantifier state, the resultant formula state is merged into φ. This process is reflected in the FORMULARELEVANT procedure below.

```
procedure FORMULARELEVANT($\varphi, S_{ctx}$)
    $qstates' \leftarrow [\,]$
    $relevant \leftarrow S_{ctx} \cap \varphi.S_{visible} \neq \emptyset$
    for $\omega \in \varphi.qstates$ do
        $r \leftarrow$ TRYEXPAND($\omega, S_{ctx}$)
        # expansion may create a new formula state $\phi_{hidden}$
        if SOME($\phi_{hidden}$) $= r$ then
            # a trigger matches; merge $\phi_{hidden}$ with $\varphi$
            $qstates' \leftarrow qstates' + \phi_{hidden}.qstates$
            $\varphi.S_{visible} \leftarrow \varphi.S_{visible} \cup \phi_{hidden}.S_{visible}$
            $relevant \leftarrow$ true
        else
            # no match; no new formula state created
            $qstates' \leftarrow qstates' + [\omega]$  # keep the quantifier state
    $\varphi.qstates \leftarrow qstates'$
    return $relevant$
```

The main procedure for this version of SHAKE is shown in Alg. 2. Its structure is almost identical to the naive version, but it uses FORMULARELEVANT to determine the relevance of each axiom in the context. A more subtle detail is that SHAKE must revisit all of the axioms in each round, as an axiom's nested quantifiers may be expanded in later rounds. Moreover, the formula state from the goal φ_0 is also part of the main loop. This way the quantifiers in the goal are also lazily expanded.

C. SHAKE *with Frequent Symbols*

Thus far we have used the symbol set abstraction introduced in §IV-A, where we exclude certain basic symbols, such as the built-in SMT-LIB functions, based on the intuition that such prevalent symbols provide little indication of relevance. We now further refine the symbol-set abstraction to reflect this intuition.

Algorithm 2 Refined SHAKE with Quantifiers

```
procedure QUANTIFIERSHAKE($\Gamma = \{\varphi_0, ..., \varphi_n\}$)
    for $\varphi_i \in \Gamma$ do
        # create the formula state
        INITFSTATE($\varphi_i$)
    # assuming $\varphi_0$ is the goal
    $S_{ctx} \leftarrow \varphi_0.S_{visible}$
    $dists, round \leftarrow \{(\varphi_0 : 0)\}, 1$
    repeat
        $acc \leftarrow \emptyset$
        for $\varphi_i \in \Gamma$ do
            $S_{prev} \leftarrow \varphi_i.S_{visible}$
            # possibly expand quantifiers
            if FORMULARELEVANT($\varphi_i, S_{ctx}$) then
                if $\varphi_i \in$ UNREACHED($dists, \Gamma$) then
                    $dists \leftarrow dists \cup \{(\varphi_i : round)\}$
                # update with previous symbols in $\varphi_i$
                $acc \leftarrow acc \cup S_{prev}$
        $S_{ctx} \leftarrow acc \cup S_{ctx}$
        $round \leftarrow round + 1$
    until ISFIXEDPOINT($dists$)
    $max\_dist \leftarrow round + 1$
    for $\varphi_i \in$ UNREACHED($dists, \Gamma$) do
        $dists \leftarrow dists \cup \{(\varphi_i : max\_dist)\}$
    return $dists$
```

In some verification languages, the SMT encoding uses certain symbols pervasively. For example, the function symbol `ApplyTT` is ubiquitous in F* queries. This is expected, as F* is based on dependent types, where terms are proofs, and `ApplyTT` represents term application. However, symbols like `ApplyTT` cause SHAKE to quickly saturate, absorbing many axioms when added to the reached symbol set.

To address this issue, we propose a simple heuristic. We define the frequency of a symbol x to be the ratio of formulas in $\Gamma = \{\varphi_0, ..., \varphi_n\}$ containing x in their symbol set:

$$freq(x) = \frac{|\{\varphi_i \mid \varphi_i \in \Gamma \land x \in \textsc{Symbols}(\varphi_i)\}|}{|\Gamma|}$$

Given a threshold θ, SHAKE excludes all symbols x such that $freq(x) > \theta$, treating them as if they were built-in functions. As a side note, this idea is related to *inverse document frequency* in information retrieval [35]. This simple approach improves pruning on certain F* queries, as we show in the evaluation.

D. SHAKE *with Distance Limit*

SHAKE is similar to *iterative deepening* [36] in spirit. However, SHAKE does not explicitly or implicitly construct a graph. Instead, SHAKE creates "layers" of axioms at different distances. By default, SHAKE runs until a fixed point, dropping axioms that are unreachable at the last layer.

SHAKE's complexity is therefore $O(DN)$, where D is the maximum distance and N is the number of axioms. In practice, our evaluation shows that D is almost always a constant $\leq$

20, while N can be in the thousands, as shown in Figure 1. SHAKE's approach improves efficiency, since a graph-based approach would take $O(N^2)$ time just to construct the graph.

Stopping SHAKE early can also be useful: by setting a distance limit, SHAKE potentially prunes even more irrelevant axioms. However, the other side of the coin is that a shallow distance limit may miss out on relevant axioms that are necessary to the goal.

The choice of distance limit thus appears to present a dilemma. However, we argue that SHAKE can leverage an unsat core as an oracle for nearly-optimal distance: since our main goal is to improve stability, we assume that an initial version of procedure P verifies, and a subsequent version P' may fail due to minor changes. Therefore, we can use the distance limit from the unsat core of P to inform the subsequent runs of P'.

In practice, we envision saving SHAKE's distance limit with source-level annotations. For example, in Dafny, a commonly used attribute is {`:timeLimit N`}, which allows the user to provide a procedure-specific time limit, overriding the default. Related attributes include {`:rlimit N`} and {`:timeLimitMultiplier X`}, which are also solver configurations. Similar annotations also exist in languages like F* and Verus [37].

SHAKE can be configured in a similar way, where the distance value is a procedure attribute. With a fresh procedure (query), the attribute is not present yet, and the solver runs as normal. If verification succeeds, we store the maximum core distance as an attribute. The next time the same procedure is verified, SHAKE uses the stored distance limit and prunes the context accordingly. Small changes in the procedure (e.g., renaming a variable) will have no impact on SHAKE's layering, and the stored limit should still work.

V. EVALUATION

In this section, we evaluate the effectiveness of SHAKE. We describe the experimental setup in §V-A. We show the distribution of distance values produced by SHAKE in §V-B. We then evaluate SHAKE's improvement of context relevance in §V-C and stability in §V-D. We further assess the impact of ignoring frequent symbols in §V-E. Lastly, in §V-F, we evaluate SHAKE's impact on solving performance in terms of run time and number of queries solved.

A. Experimental Setup

In the evaluation, we run SHAKE in two different modes.

- **Default Mode:** SHAKE computes the distances and then prunes the unreachable axioms, i.e., axioms in the last layer discussed in §IV-A.
- **Oracle Mode:** We obtain an "ideal" distance by employing the unsat core as an oracle. We then use SHAKE to prune axioms beyond the oracle distance.

To evaluate stability, we use SHAKE's oracle mode. As discussed in §IV-D, to counter instability, we assume a prior working version of the query that produces a core, from which we obtain the oracle distance.

To evaluate standard solving performance overhead, i.e., without any query mutation, we use the oracle mode along with the default mode. This provides a best-case and worst-case comparison for SHAKE's performance impact as a preprocessor.

By default, SHAKE does not ignore any query-defined symbols based on their frequencies (§IV-C). We only experiment with frequency configuration in §V-E.

We use the default settings for Mariposa [32], including a time limit of 60 seconds for each query. We experiment with recent versions of two SMT solvers, Z3 version 4.12.5 and cvc5 version 1.1.1. We conduct our experiments on machines with an Intel Core i9-9900K (max 5.00 GHz) CPU, 128 GB of RAM, and the Ubuntu 20.04.3 LTS operating system.

B. Distribution of SHAKE *Distances*

First, we evaluate how well SHAKE distances reflect the relevance of axioms. Recall that for an original query $\Gamma = \{\varphi_0, ..., \varphi_n\}$, SHAKE computes the distances:

$$dists = \{(\varphi_0 : d_0), ..., (\varphi_n : d_n)\}$$

Let $\Gamma_c \subseteq \Gamma$ be the core provided by the solver. We can then determine the maximum distances for the original query and the core:

$$\begin{aligned} d_{orig} &= \max(d_i \mid (\varphi_i : d_i) \in \Gamma) \\ d_{core} &= \max(d_i \mid (\varphi_i : d_i) \in \Gamma_c) \end{aligned}$$

Intuitively, if $d_{orig} > d_{core}$, then SHAKE is able to differentiate between core and non-core axioms: the more significant the difference is, the more we can safely prune layers in between with no loss of core axioms.

As shown in Figure 5-Figure 9, the maximum distances are upper-bounded by 20 for all queries from the five projects in this study. Moreover, there is usually a clear difference between d_{orig} and d_{core}. As an example, Figure 5 shows the distributions from Komodo$_D$. Note the strong separation between the two: the median d_{core} is 2, while the median d_{orig} is 8. Moreover, the distribution of the d_{core} is light-tailed, where a distance of 3 covers almost the entirety of the query set.

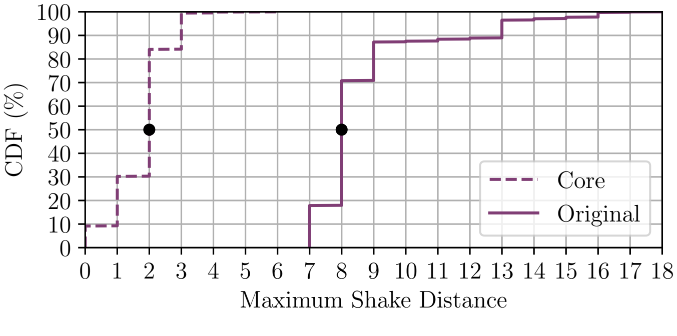

Fig. 5. **Maximum SHAKE Distances for Komodo$_D$.** There is a clear separation between the distance values of core axioms versus original axioms.

However, in Figure 9, we observe that vWasm$_F$ is a bit of an outlier (again). As we discussed in §III-C, the vWasm$_F$

query set starts off with much higher context relevance; thus we do not expect much room for differentiation using SHAKE's distance.

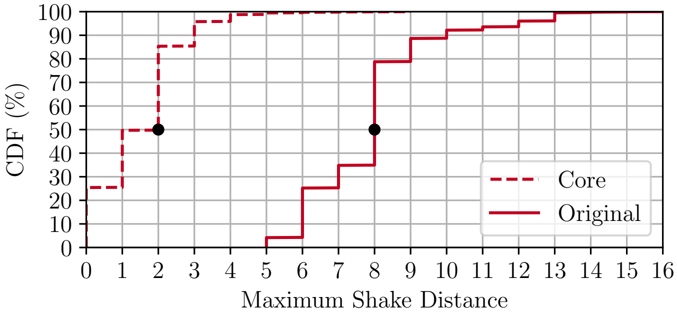

Fig. 6. **Maximum SHAKE Distances for VeriBetrKV$_L$.**

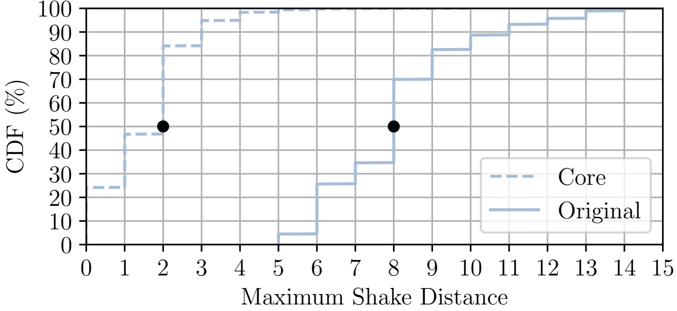

Fig. 7. **Maximum SHAKE Distances for VeriBetrKV$_D$.**

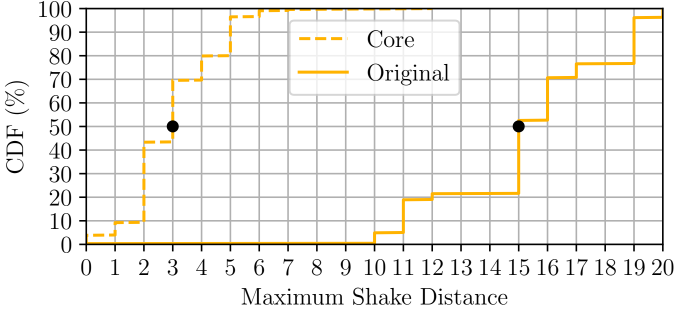

Fig. 8. **Maximum SHAKE Distances for DICE$^{\star}_F$.**

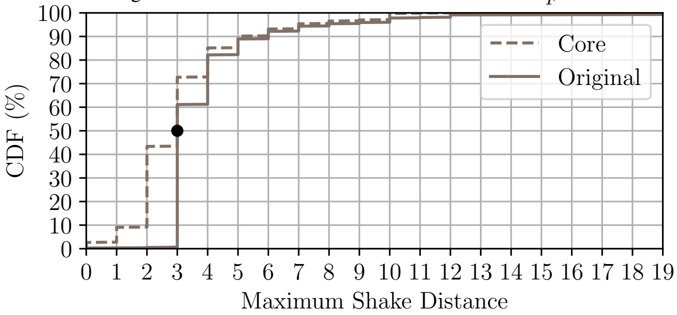

Fig. 9. **Maximum SHAKE Distances for vWasm$_F$.**

C. *Context Relevance Ratio*

Now that we have demonstrated that SHAKE differentiates core and non-core axioms, we evaluate how much context pruning SHAKE enables. Since our main goal is to mitigate instability, we run SHAKE in oracle mode. As in §III-B, we compute the relevance ratio of the pruned query:

$$\frac{\#\text{ core axioms} + 1}{\#\text{ axioms after pruning} + 1} \times 100\%$$

In Figure 10, we present the relevance ratios that SHAKE achieves. We see significant improvements over the original queries as shown in Figure 2. For example, in VeriBetrKV$_L$, the median relevance ratio (MRR) is 0.32% in the original queries, while the MRR increases to 3.46% with oracle SHAKE. Overall, SHAKE improves the MRR by 3–10×. We note the intersection on the right side of the plot, where the relevance ratio is 100%. In those cases, SHAKE matches the unsat core when only given the oracle distance.

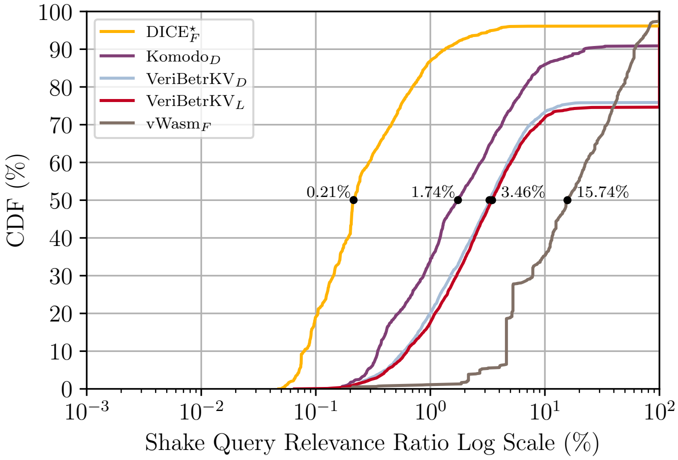

Fig. 10. **Oracle SHAKE Query Context Relevance.** Oracle SHAKE shows improvement of context relevance over Figure 2 by 3–10×.

D. *Stability Improvement*

Next, we evaluate if the improved context relevance translates into improved stability. We assess stability in the same way as in the unsat core experiments in §III-C, both from a preservation and mitigation perspective.

In Figure 11, we report the stability scores for oracle SHAKE on Z3 version 4.12.5. We include all of the unstable queries found in the original Mariposa query set (just as we did in Figure 3), and then we sample roughly the same number of stable queries (110 from each project). We observe that SHAKE generally preserves stability, and achieves reasonable success mitigating instability, with an overall mitigation score of 29.7%. We also see that the naive SHAKE from §IV-A performs much worse, achieving an overall mitigation score of only 11%.

We observe that DICE$^{\star}_F$ sees much less mitigation. We attribute this to F*'s pervasive use of certain function symbols (such as `ApplyTT`) in its query encoding. In §V-E, we evaluate the effectiveness of suppressing such symbols based on their frequency. We also observe that SHAKE does not help with the unstable queries in vWasm$_F$. Since the unsat core is not effective on vWasm$_F$, this is unsurprising.

To further validate the stability improvement, we also evaluate SHAKE with cvc5 version 1.1.1. However, cvc5

Project	Original Count		Oracle Naive SHAKE		Oracle SHAKE	
	Stable	Unstable	Preservation	Mitigation	Preservation	Mitigation
Komodo$_D$	110	93	99.1%	7.5%	100.0%	25.8%
VeriBetrKV$_D$	110	172	100.0%	12.2%	98.2%	23.3%
VeriBetrKV$_L$	110	256	100.0%	11.7%	100.0%	37.9%
DICE$^\star_F$	110	20	100.0%	10.0%	100.0%	5.0%
vWasm$_F$	110	4	100.0%	0.0%	96.4%	0.0%
Overall	550	545	99.8%	11.0%	98.9%	29.7%

Fig. 11. **Oracle SHAKE Query Stability on Z3 4.12.5.** We include oracle naive SHAKE (middle column) from §IV-A for comparison. Oracle SHAKE, which employs the quantifier handling strategy from §IV-B, shows similar preservation, but stronger mitigation results.

is known to not work well with queries from Dafny and $F^\star$, as acknowledged by cvc5's developers [15]. In fact, to make this evaluation possible, we had to first syntactically transform the queries into a format cvc5 could parse. Even then, cvc5 times out on many of the original queries (whereas Z3 succeeds for nearly all of them). Hence, we only evaluate the stability of original queries that do not timeout with cvc5. This necessarily introduces bias in the resulting query sample, so the stabilization results from Z3 and cvc5 should not be directly compared.

With that caveat in mind, we present the stability scores for oracle SHAKE on cvc5 in Figure 12. Generally, the preservation scores are quite strong. The overall mitigation score of 41.3% is promising as well.

Project	Original Count		Oracle SHAKE	
	Stable	Unstable	Preservation	Mitigation
Komodo$_D$	110	36	100.0%	41.7%
VeriBetrKV$_D$	110	143	94.5%	48.3%
VeriBetrKV$_L$	110	210	100.0%	37.1%
DICE$^\star_F$	110	17	100.0%	100.0%
vWasm$_F$	110	27	99.1%	0.0%
Overall	550	433	98.7%	41.3%

Fig. 12. **Oracle SHAKE Query Stability on cvc5 1.1.1.**

E. *Frequency Configuration*

As discussed in §IV-C, SHAKE can optionally take in a threshold θ and ignore any symbol x such that $\mathit{freq}(x) > \theta$. We now evaluate if this configuration can help with stability. Intuitively, if θ is set properly, SHAKE can ignore trivial matches due to pervasively used symbols. However, if θ is too low, SHAKE may not reach axioms that are actually relevant, e.g., the ones in the core.

We continue to use the oracle mode for this experiment. Recall that SHAKE assigns the unreachable axioms to the maximum distance. When core axioms end up being unreachable, oracle SHAKE cannot safely prune any axioms, since this could introduce incompleteness. Therefore, in addition to the mean relevance ratio (MRR), we also report the *fallback rate* (FR), which is the percentage of queries where oracle SHAKE cannot prune any axioms.

First, we discuss the choice of θ with an experiment on query relevance. $\theta = 1.00$ means no symbols are pruned based on frequency. In Figure 13, we observe that there is a trade-off between the relevance ratio and the fallback rate. For example, in Komodo$_D$, $\theta = 0.15$ achieves the highest MRR, but also has the highest FR. In vWasm$_F$, since the context starts with high MRR, lower θ values only increase FR. In general, $\theta = 1.00$ (no frequency pruning) tends to balance the two metrics.

		Orig.	$\theta = 1.00$	$\theta = 0.30$	$\theta = 0.15$
Komodo$_D$	MRR	0.57	1.74	1.74	2.40
	FR	–	0.39	6.08	13.14
VeriBetrKV$_D$	MRR	0.33	3.28	3.35	2.51
	FR	–	1.45	5.74	28.49
VeriBetrKV$_L$	MRR	0.32	3.46	3.59	3.03
	FR	–	1.42	5.45	15.91
DICE$^\star_F$	MRR	0.06	0.21	0.32	0.88
	FR	–	4.44	5.90	7.10
vWasm$_F$	MRR	3.76	15.74	16.0	16.22
	FR	–	5.99	6.11	12.51

Fig. 13. **Oracle SHAKE Context Relevance with Frequency Configuration.** Higher MRR means more relevant context. Higher FR means more queries for which oracle SHAKE does not prune any axioms.

However, for DICE$^\star_F$, the results indicate that $\theta = 0.15$ is a promising setting, since the MRR is increased by 4× with respect to $\theta = 1.00$, while sacrificing three percentage points of FR. We test the stability of using $\theta = 0.15$ on DICE$^\star_F$ with Z3 and find that it improves stability by 6× compared to oracle SHAKE with $\theta = 1.00$.

F. *Solving Performance Impact*

Proof instability is a pernicious problem in program verification, so it might be reasonable to expect developers to be willing to trade worse solving performance for greater stability. Fortunately, our results show that such a trade is largely unnecessary: SHAKE adds relatively little overhead and even improves performance in some cases.

To evaluate solving performance, for each solver (Z3 and cvc5), we compare the following three scenarios.

- **Baseline.** The original queries are directly given to the solver.
- **Default SHAKE.** The queries are preprocessed by SHAKE in default mode and then given to the solver.
- **Oracle SHAKE.** The queries are preprocessed by SHAKE in oracle mode and then given to the solver.

Since SHAKE is a preprocessor, its runtime includes the time spent on computing the distances and the time spent in IO. When reporting the runtime, we exclude the latter, since we expect SHAKE to eventually be incorporated directly into solvers, where parsing is already being done. Therefore, the runtime for the SHAKE modes is the time spent on computing the distances plus the time spent by the solver on the pruned queries. Each query is given a 60 second timeout, so if SHAKE distance computation and solver together takes more than that, the query is not considered solved.

First we present the number of queries solved in each scenario in Figure 14. Generally SHAKE adds a minor overhead to Z3, but sometimes solves a few more in oracle mode. However, if we consider cvc5, SHAKE usually improves the number of queries solved, even in default mode. Notably, in DICE$^{\star}_{F}$, cvc5 solves 259 queries in the baseline; even with default SHAKE, it solves 190 more (+79%); with oracle SHAKE, it solves 424 more (+163%).

	Solver	Baseline	Default	Oracle
Komodo$_D$	Z3 cvc5	1,983 342	-0.10% +1.75%	+0.30% +21.64%
VeriBetrKV$_D$	Z3 cvc5	5,103 2,571	-0.78% +9.14%	-0.61% +20.77%
VeriBetrKV$_L$	Z3 cvc5	5,167 3158	-0.41% +8.90%	-0.04% +13.01%
DICE$^{\star}_{F}$	Z3 cvc5	1,493 259	-0.07% +73.36%	+0.33% +163.71%
vWasm$_F$	Z3 cvc5	1,733 1,630	-0.29% -0.12%	-0.35% -0.12%
Overall	Z3 cvc5	15,479 7,960	-0.45% +8.92%	-0.18% +18.10%

Fig. 14. **Queries Solved with SHAKE as a Preprocessor.**

To present the runtime performance, we use survival plots; Brain et al. [38] provide a detailed explanation, but in short, a survival plot shows the cumulative number of queries solved within a total time budget. Therefore, a curve that is higher and to the left indicates better performance.

In each plot, we show six curves, based on the three scenarios for each of the two solvers. Generally, SHAKE adds a minor overhead to Z3, but often improves the solving speed on cvc5. For example, in Figure 16, we show the survival plot for VeriBetrKV$_D$. SHAKE's impact on Z3 is almost negligible, whether in default or oracle mode. However, for cvc5, SHAKE does improve on the solving speed, as well as the number of queries solved, not only in oracle mode, but also in default mode. In Figure 17, VeriBetrKV$_L$ shows a similar trend as in VeriBetrKV$_D$.

In Figure 18, we show the results for DICE$^{\star}_{F}$. We observe that default SHAKE adds a minor overhead to Z3, but oracle SHAKE has little impact. On cvc5, as we discussed earlier, SHAKE significantly improves the number of queries solved and improves the runtime as well.

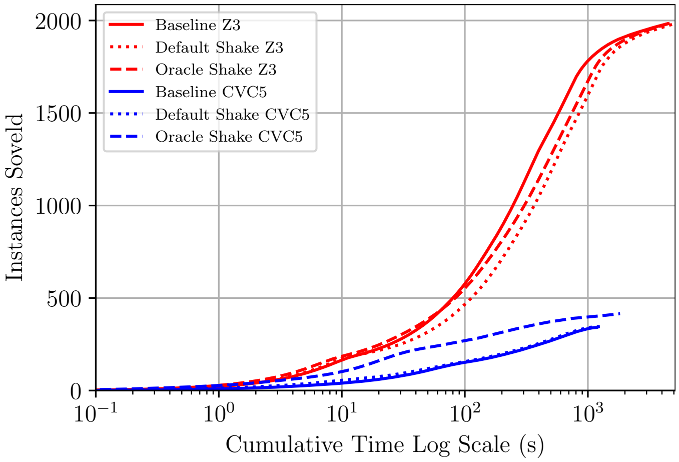

Fig. 15. **SHAKE Performance Survival Plot for Komodo$_D$.**

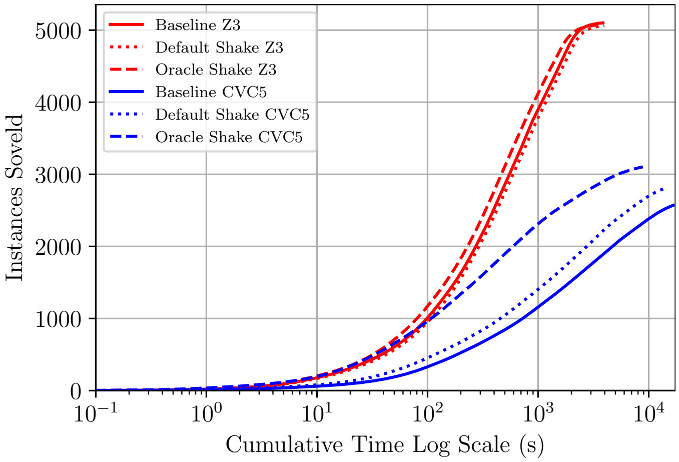

Fig. 16. **SHAKE Performance Survival Plot for VeriBetrKV$_D$.**

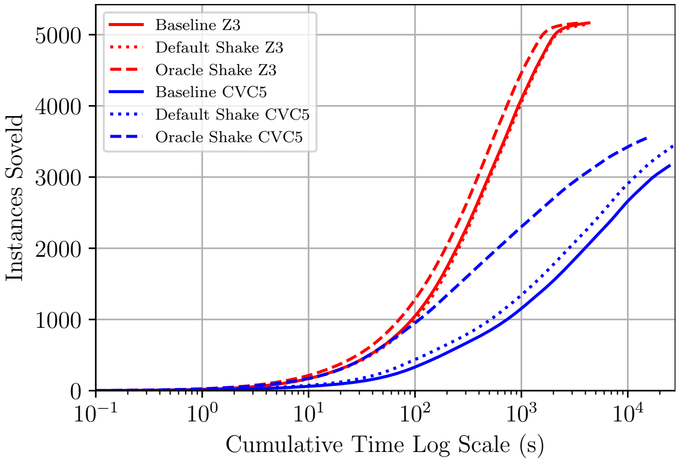

Fig. 17. **SHAKE Performance Survival Plot for VeriBetrKV$_L$.**

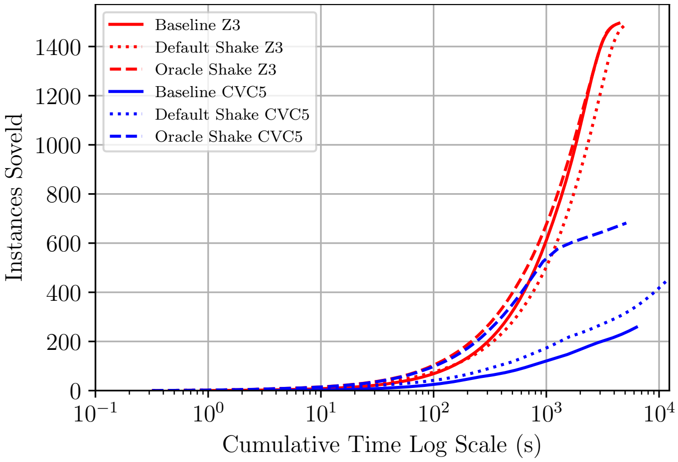

Fig. 18. **SHAKE Performance Survival Plot for DICE$^{\star}_{F}$.**

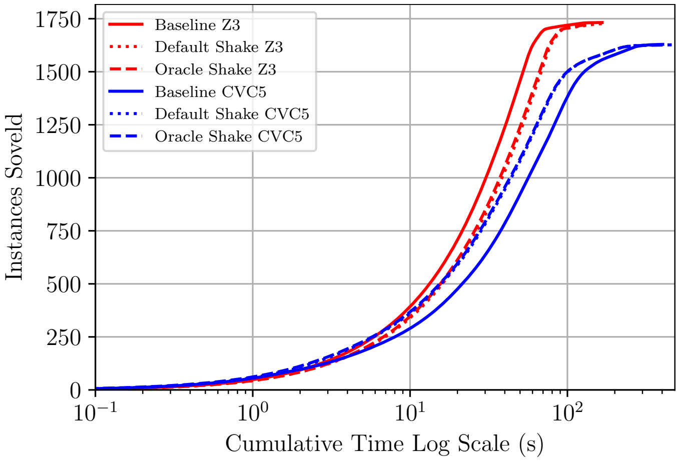

Fig. 19. **SHAKE Performance Survival Plot for vWasm$_{F}$.**

VI. Related Work

The problem of proof instability in the context of program verification has been a long standing issue. For example, Hawblitzel et al. bemoan the instability of certain SMT queries [19], and the Komodo authors describe proof instability as "the most frustrating recurring problem" [18].

The Mariposa project [15] is the first effort to quantify in a statistically rigorous way the instability of SMT queries with respect to a solver. The authors measure instability in six large-scale verification projects across eight SMT solver versions. However, their focus is on quantifying instability, rather than understanding or mitigating it.

Our SHAKE technique resembles an algorithm first implemented in the Sumo INference Engine [39]. The Sine algorithm selects relevant axioms in automated theorem proving (ATP) problems [24]. As in SHAKE, Sine uses overlapping symbols to iteratively determine relevance. A similar strategy was later employed by the lightweight relevance filtering algorithm [40]. However, these algorithms target ATP problems, e.g., those from TPTP [41], which usually covers domains outside those in the SMT queries produced by program verification. Moreover, a major difference between SHAKE and these algorithms is the strategy SHAKE employs to handle quantified expressions. In SHAKE, we make use of quantifier patterns and perform lazy quantifier expansion, which is not present in the earlier algorithms.

VII. Limitations

This work has several limitations. First, we have only studied verification projects written in Dafny and F*, which do not necessarily represent the entire spectrum of automated program verification. For example, we have excluded Mariposa's Komodo$_{S}$, since it is restricted to the decidable fragments of SMT and does not fit our description of VCG in §III-D. Second, unsatisfiable cores have guided much of our analysis and experiments, but the solver-produced core is not a perfect oracle of relevant assertions. For example, the solver makes no guarantee about the minimality (necessity) of the core assertions. Third, our proposed technique, SHAKE, needs to assume an oracle distance limit and/or a frequency threshold to be effective. While the assumption of oracle configurations can be met when dealing with unstable queries, ideally we would like to remove this dependency, possibly by integrating SHAKE into the SMT solver in future work. Lastly, SHAKE works at the SMT level, and thus may have less precision compared to VCG-level pruning. Nevertheless, SHAKE demonstrates the general applicability of context pruning to improve stability, and we leave language-specific adaptations to future work.

VIII. Conclusion

In this work, we empirically study the problem of proof instability in SMT-based program verification. We find that irrelevant context is a major source of instability. We then propose SHAKE, a novel SMT-level context pruning algorithm as a mitigation technique. We demonstrate that SHAKE can improve the stability of automated program verification using queries from real-world projects. Furthermore, we show that SHAKE can potentially improve standard SMT-solving performance on these queries as well. We hope our work offers useful insights into the phenomenon of instability and the connection between automated program verification and theorem proving.

IX. Acknowledgement

Chris Hawblitzel and Doug Woos worked on a prototype algorithm for SMT-level tree-shaking in 2016. This work is a redesign and extension of that effort. We thank Haniel Barbosa and Livia Sun for their advice on cvc5 configuration; Jialin Li for her suggestion of storing distance limit as a procedure attribute; and the anonymous reviewers for their helpful feedback on the paper.

This work was supported in part by the National Science Foundation (NSF) under grant 2224279, funding from AFRL and DARPA under Agreement FA8750-24-9-1000, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab).

References

[1] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., 2010.

[2] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, “Dependent Types and Multi-Monadic Effects in F*,” in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[3] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM*, vol. 12, no. 10, 1969.

[4] E. W. Dijkstra, “Guarded Commands, Nondeterminacy and Formal Derivation of Programs,” *Commun. ACM*, aug 1975.

[5] C. Barrett, A. Stump, C. Tinelli *et al.*, “The SMT-lib Standard: Version 2.0,” in *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2010.

[6] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, “cvc5: A Versatile and Industrial-Strength SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.

[8] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel, “Linear Types for Large-Scale Systems Verification,” in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2022.

[9] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin, “HACLxN: Verified generic SIMD crypto (for all your favorite platforms),” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2020.

[10] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, “HACL*: A verified modern cryptographic library,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[11] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin, “EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[12] Y. Zhou, S. Gibson, S. Cai, M. Winchell, and B. Parno, “Galápagos: Developing verified low-level cryptography on heterogeneous hardware,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2023.

[13] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, “EverParse: Verified secure Zero-Copy parsers for authenticated message formats,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019.

[14] J. Bosamiya, W. S. Lim, and B. Parno, “Provably-Safe Multilingual Software Sandboxing using WebAssembly,” in *Proceedings of the USENIX Security Symposium*, August 2022.

[15] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, “Mariposa: Measuring SMT instability in automated program verification,” in *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, October 2023.

[16] A. Tomb and J.-B. Tristan, “Avoiding verification brittleness in Dafny,” https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/, 2023.

[17] M. Dodds, “Formally Verifying Industry Cryptography,” *IEEE Security and Privacy Magazine*, vol. 20, no. 3, 2022.

[18] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[19] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, “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.

[20] K. R. M. Leino and C. Pit-Claudel, “Trigger Selection Strategies to Stabilize Program Verifiers,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, S. Chaudhuri and A. Farzan, Eds., 2016.

[21] J. W. Cutler, E. Torlak, and M. Hicks, “Improving the stability of type soundness proofs in Dafny,” in *Proceedings of the First Workshop on Dafny*, 2024.

[22] S. Ho and C. Pit-Claudel, “Incremental proof development in Dafny with module-based induction,” in *Proceedings of the First Workshop on Dafny*, 2024.

[23] S. McLaughlin, G.-A. Jaloyan, T. Xiang, and F. Rabe, “Enhancing proof stability,” in *Proceedings of the First Workshop on Dafny*, 2024.

[24] M. Fitting, *First-order Logic and Automated Theorem Proving*. Springer Science & Business Media, 2012.

[25] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, “IronFleet: Proving Practical Distributed Systems Correct,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[26] A. Arasu, T. Ramananandro, A. Rastogi, N. Swamy, A. Fromherz, K. Hietala, B. Parno, and R. Ramamurthy, “FastVer2: A provably correct monitor for concurrent, key-value stores,” in *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)*, January 2023.

[27] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, “Storage Systems are Distributed Systems (So Verify Them That Way!),” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[28] T. Hance, Y. Zhou, A. Lattuada, R. Achermann, A. Conway, R. Stutsman, G. Zellweger, C. Hawblitzel, J. Howell, and B. Parno, “Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2023.

[29] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran *et al.*, “Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3,” in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[30] B. Cook, “Formal reasoning about the security of amazon web services,” in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 38–47.

[31] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta, “Hardening Attack Surfaces with Formally Proven Binary Format Parsers,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2022. [Online]. Available: https://www.fstar-lang.org/papers/EverParse3D.pdf

[32] “Mariposa Public Repository,” https://github.com/secure-foundations/mariposa, accessed: May 2023.

[33] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1980, aAI8011683.

[34] M. Moskal, “Programming with triggers,” in *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2009.

[35] J. Ramos *et al.*, “Using tf-idf to determine word relevance in document queries,” in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[36] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[37] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, “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.

[38] M. Brain, J. H. Davenport, and A. Griggio, “Benchmarking solvers, SAT-style.” in SC^2@ *ISSAC*, 2017.

[39] K. Hoder and A. Voronkov, “Sine qua non for large theory reasoning,” in *International Conference on Automated Deduction*. Springer, 2011, pp. 299–314.

[40] J. Meng and L. C. Paulson, “Lightweight Relevance Filtering for Machine-Generated Resolution Problems,” *Journal of Applied Logic*, vol. 7, no. 1, pp. 41–57, 2009.

[41] G. Sutcliffe and C. Suttner, “The TPTP Problem Library,” *Journal of Automated Reasoning*, vol. 21, pp. 177–203, 1998.