Formal Methods in Computer-Aided Design 2023

Mariposa: Measuring SMT Instability in Automated Program Verification

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

***Abstract*—Program verification has been successfully applied to increasingly large and complex systems. Much of this recent success can be attributed to the automation provided by dispatching verification condition queries via SMT solvers. However, multiple teams anecdotally report that this style of automated verification is plagued by *proof instability*, where semantically irrelevant changes to the query can have large effects on the SMT solver's response.**

In this work, we present Mariposa, a tool to detect and quantify instability. To better understand the status quo of instability, we apply Mariposa to a set of 17,043 SMT queries from six existing program verification projects. We discover that SMT solver upgrades often make projects less stable, and that the most recent SMT solver version is unstable on 2.6% of the queries. For individual projects, the unstable ratio can grow to 5.0%. Based on our experimental results, we curate the Mariposa benchmark, which we hope will help measure and incentivize stability improvements in SMT-based program verification.

I. Introduction

Software verification can statically guarantee a program's correctness, reliability, and/or security. In recent years, we have seen significant progress scaling software verification up to large, practical programs, both in academia [1–8] and industry [9–11].

Much of this success relies on Satisfiability Modulo Theories (SMT) solvers [12–15]. The developer writes specifications, proofs, and code, which are transformed into a verification condition [16], expressed as a query in the SMT-LIB [13] format. The SMT solver then does the heavy lifting by checking the verification condition, essentially verifying that the code meets its specification. In practice, this process is iterative: when a query fails, the developer adjusts the specifications, proofs, and/or code until the updated query is accepted and the developer moves on to the next code region.

Unfortunately, automated program verification suffers from *proof instability* [11, 17, 18], where seemingly *irrelevant* changes to the verification condition can cause notable variation in SMT solver performance. For example, simply renaming a source-level variable may cause a verified procedure to take orders of magnitude longer to verify, or even to fail to verify at all. In either case, the developer must tediously supply additional proof hints that attempt to steer the SMT solver back towards a fast and successful verification result.

Instability poses a significant challenge for large-scale, industrial-level program verification. Concretely, in the verification projects we study, we find up to 5% of queries to be unstable with the most recent SMT solver version (Section IV-D). For developers, such instability disrupts their iterative workflow, as it substantially lengthens their code-prove-debug cycle. Moreover, spurious failures may require developers to fix issues that arise in code or proofs they did not write and may not even understand. In a large team of developers, this problem is amplified, as independent and concurrent changes to the codebase potentially create instability that is only visible after changes are merged. In short, instability impedes monotonic progress in developing a verified codebase.

While the program verification community has recognized the issue of instability [11, 17, 18], popular automated verification tools like Dafny [19] and F* [20] only offer heuristic options to identify it [21, 22]. Furthermore, our results show that these heuristics only capture a fraction of the problem (Section IV-D).

In the SMT community, SMT-COMP [23], the annual competition for SMT solvers, does not include any benchmarks for evaluating stability. Possibly as a result, the stability of some program verification projects actually *deteriorates* with solver upgrades (Section IV-D).

We believe there is a need for a systematic study of the instability phenomenon, where concrete data and statistical analysis can inform both the program verification and SMT communities. A robust measurement methodology can help program verification frameworks adapt their query-generation strategy to avoid issuing unstable queries. For SMT solvers, a benchmark for measuring instability would help evaluate strategies for mitigating it, as well as help prevent stability regressions. In this work we fill this need with the following contributions.

- We present a methodology (and a concrete tool named Mariposa[1]) to detect and quantify SMT-based proof instability (Section III).

[1]Mariposa is Spanish for butterfly. The name is inspired by the butterfly effect, where small changes can have large effects.

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26

- We perform a detailed empirical study analyzing the (in)stability of six projects written in three program verification frameworks across multiple SMT solver versions (Section IV). The study generates over three million SMT queries that consume ~ 578 CPU days.
- We distill our study's queries into the Mariposa benchmark to facilitate future research on measuring and mitigating instability (Section V).
- Our study quantifies anecdotal reports of SMT instability, showing that it affects a non-trivial number of queries and often grows worse with new solver versions. We also find that multiple mutation methods are needed to uncover unstable proofs.

The SMT queries and results from our experiments, the source code for the Mariposa tool, and the Mariposa benchmark are all publicly available: https://github.com/secure-foundations/mariposa.

II. Related Work

To the best of our knowledge, the problem of SMT proof instability was first reported by the developers of Ironclad Apps [17], who noticed instability in certain non-linear integer arithmetic queries. In the later Komodo work [24], instability was described as "the most frustrating recurring problem." More recently, Galois highlighted the "fragility of proofs" as a challenge in formally verified industry cryptography [11].

Leino and Pit-Claudel studied the problem of SMT instability in the specific context of Dafny quantifier instantiation [18]. They investigated trigger loops as a possible source of instability, improved algorithms for trigger selection, and then used ad hoc instability measures to evaluate the impact of their algorithms.

The SAT Competition [25] and SMT-COMP [23] may perform benchmark scrambling before evaluating the solvers' performance. Scrambling involves syntactic transformations similar to our query mutations (Section III). However, scrambling is not sufficient (nor intended) to characterize stability. Prior work has examined the impact of scrambling on competition results [26, 27].

Most work on testing SMT solvers focuses on finding unsoundness bugs [28–33]. One exception is Janus [34], which finds incompleteness bugs, where a query unexpectedly returns `unknown`, placing it closer to our work. However, Janus does not offer a metric for instability, nor does it target program verification queries.

III. Methodology

In this section, we outline our methodology for characterizing proof instability. At a high level, our goal is to answer two main questions for a given query Q and solver S: (1) Is Q stable or unstable under S? and (2) How stable or unstable is it?

Intuitively, instability means that the performance of S diverges when seemingly irrelevant mutations are applied to Q. Our methodology, detailed below, follows this intuition. First, we characterize the queries of interest, drawn from prior program verification projects (Section III-A). Next, we describe the mutations chosen for our study and the rationales behind the choices (Section III-B). We then propose a scheme to differentiate stable and unstable queries (Section III-C), addressing question (1) above. Finally, we elaborate on metrics used to quantify stability (Section III-F), addressing question (2).

A. Characterizing Program Verification Queries

This study focuses on queries from automated verification projects, where instability is problematic. Here, we describe their general characteristics, which might differ from those in other domains, such as symbolic execution or model checking. We discuss the specific verification projects chosen for our study in Section IV-B.

Relevant Logics. Program verification queries involve a mixture of bit-vector, integer arithmetic, and uninterpreted functions, typically with quantifiers. There is no single SMT-LIB logic (e.g. `QF_UF` or `NIA`) that captures these at the same time, and thus program verification queries commonly use the `ALL` logic.

Expected Query Result. The goal of program verification is to prove that a property holds in all cases. Therefore, the SMT query is formulated as the *negation* of the desired property, such that a successful proof is indicated via an `unsat` result. Intuitively, if the result is `sat`, then the property is violated in at least one case (the satisfying assignment). For this study, the expected result is always `unsat`, which means that the property holds in all cases (i.e., the program verifies).

Expected Response Time. As discussed in Section I, the process of developing verified software is iterative. Given that the developer is blocked while the solver is running, the solver's run time should be in the responsive range of human interaction. For most of the projects in our study, the solver time limits used during development are under 30 seconds.

B. Mutation Methods

In this study, we focus on mutation methods that yield queries that are both *semantically equivalent* and *syntactically isomorphic*; i.e., the original query Q and its mutated version Q' share the same semantic meaning and syntactic structures. Hence it seems reasonable to expect similar performance from the solver on both queries.

Semantic Equivalence. Q and Q' are semantically equivalent when there is a bijection between the set of proofs for Q and those for Q'. In other words, a proof of Q can be transformed into a proof of Q', and vice versa.

Syntactic Isomorphism. Q and Q' are syntactically isomorphic if there exists a one-to-one correspondence between the symbols (e.g., variables) and commands (e.g., assertions). In other words, each symbol or command in Q has a counterpart in Q', and vice versa.

For our concrete experiments, we are interested in mutations that also correspond to common developer practices. Specifically, we consider the following three mutation methods:

- **Assertion Shuffling**. Reordering of source-level lemmas or code methods is a common practice when developing verified software. Such reordering roughly corresponds to shuffling the order of commands in the generated SMT query. Specifically, SMT queries introduce constraints using the `assert` command. Shuffling the order in which the constraints are declared guarantees syntactic isomorphism. Further, the order within a local context is irrelevant to the query's semantics.
- **Symbol Renaming**. It is common to rename source-level methods, types, or variables, which roughly corresponds to α-renaming the symbols in the SMT queries. Renaming preserves semantic equivalence and syntactic isomorphism, as long as the symbol names are used consistently.
- **Randomness Reseeding**. SMT solvers optionally take as input a random seed, which is used in some of their non-deterministic choices. Changing the seed has no effect on the query's semantics but is known to affect the solver's performance. Historically, some verification tools have attempted to use reseeding to measure instability: Dafny[2] and F* have options to run the same query multiple times with different random seeds and report the number of failures encountered.

When a mutation method is exhaustively applied to a query Q, it produces a set of mutated queries M_Q, which also includes Q itself. Consider assertion shuffling as an example. If Q contains 100 assertions, then M_Q would have $100! \approx 9 \times 10^{157}$ permutations of Q. We refer to Q as the *original* query and members of M_Q as *mutants*.

C. Detecting Stability

Intuitively, stability is a performance property over M_Q. That is, whether a query-solver pair (Q,S) is stable or not depends on how the mutants perform. To simplify the discussion, we assume for now that a single mutation method, such as assertion shuffling, is used. In Section III-E, we discuss how to aggregate results from multiple mutation methods.

Mutant Success Rate. A natural performance metric is the success rate of solver S over M_Q. More precisely, it is the percentage of queries in M_Q that are proven (i.e., that return the expected `unsat` result).

[2]Dafny has recently started to perform shuffling and renaming. The option has changed from `randomSeedIterations` to `randomizeVcIterations`.

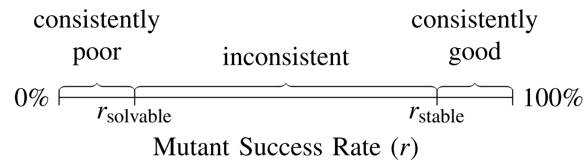

Fig. 1. **Intuition for Our Stability Categories.** (S,Q) is a solver-query pair. r is the mutant success rate. When $r < r_{\text{solvable}}$, Q is not solvable under S. When $r > r_{\text{stable}}$, Q is stable under S. Otherwise, Q is unstable under S.

The success rate, which we denote by r, reflects performance consistency. A low r indicates consistently poor results; a high r indicates consistently good results; and a moderate r indicates inconsistent results, i.e., instability. This intuition is illustrated in Figure 1.

We thus define four stability categories using the success rate r. The scheme includes two additional parameters: r_{solvable} and r_{stable}, which correspond respectively to the lower and upper bounds of the success rate range for unstable queries.

- `unsolvable`. Q is too difficult for solver S ($r < r_{\text{solvable}}$). For example, if S gives up and returns `unknown` for all members of M_Q, we may conclude that S is unable to solve Q or any version of it.
- `unstable`. S cannot consistently find a proof in the presence of mutations to Q ($r_{\text{solvable}} \leq r < r_{\text{stable}}$).
- `stable`: S proves M_Q consistently ($r \geq r_{\text{stable}}$).
- `inconclusive`: statistical tests do not result in enough confidence to draw a conclusion.

Mutant Sampling. In practice, it is often intractable to enumerate all members of M_Q (recall the 100! mutants from our shuffling example), so r is generally unknown. Therefore we use statistical tests to estimate r from a sample of mutants. We use $\hat{M}_Q$ and $\hat{r}$ to denote sample mutants and sample success rate, respectively.

Our scheme is based on comparing proportions, so we use the Z-test [35], which is a commonly used statistical test to make inferences about the true proportion of a population based on a set of samples. The test is parameterized by the alpha level, which specifies confidence in its result. We use an alpha level of 0.05 (i.e., 95% confidence), which is a standard choice.

Figure 2 shows our proposed workflow for categorizing the stability of a query-solver pair. For a statistical test (shown as a trapezium shape), if we reject the null hypothesis (H_0), there is enough confidence to conclude that the alternative hypothesis (H_A) is true. For example, in the Instability Test, if we reject H_0, we are 95% sure that H_A is true, i.e., $r < r_{\text{stable}}$. However, failing to reject H_0 simply means the result is not statistically significant. That is, failing the Instability Test does *not* imply stability. Hence, we test again using the opposite hypothesis. If the test is still not significant, we do not have a conclusive result.

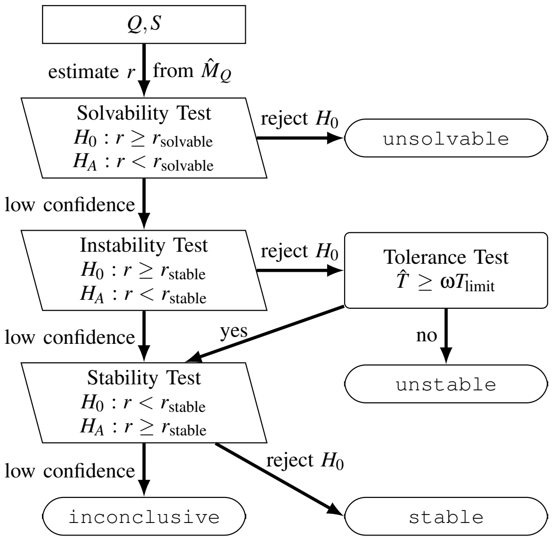

Fig. 2. **Flowchart for Stability Categorization.** We output a stability category based on the performance of the mutants though a series of hypothesis tests. An additional tolerance test is used to filter out queries that are finishing close to the time limit T_{limit}.

A natural goal to make is to pick a sufficiently large sample size such that very few cases are `inconclusive`. As a sanity check, we expect to conclude `unsolvable` ($r < r_{\text{solvable}}$) if no sample mutant succeeds ($\hat{r} = 0\%$) using an alpha level of 0.05.

We thus calculate the required sample sizes for different values of r_{solvable}. For $r_{\text{solvable}} = 1\%$, we need 269 mutants to be 95% sure that the true success rate is less than 1%. On the other hand, if $r_{\text{solvable}} = 5\%$, 60 mutants are more than enough. Similarly, we expect to conclude `stable` if the sample success rate $\hat{r}$ is 100%. We note this is symmetric to the previous scenario, and thus, to conclude `stable` ($r \geq 95\%$), 60 sample mutants all succeeding ($\hat{r} = 100\%$) is sufficient.

For our experiments, we use $r_{\text{solvable}} = 5\%$, $r_{\text{stable}} = 95\%$, and 60 mutants for each mutation method.

D. Accounting for Time Limits

Since there is no guarantee that a solver will terminate, we impose a time limit T_{limit} on all of our experiments. Solvers may allow the user to bound the solver execution with a resource limit (`rlimit`) instead of a time limit, in an effort to make results more consistent across machines with different computational abilities. However, the resource tracking often counts only some of the resources used (e.g., it may ignores resources spent inside a theory solver). Further, there is no guarantee of consistency across solver versions, let alone across different solvers. Hence, in this work Mariposa uses execution time as a more universal measure.

In the categorization scheme, a mutant that times out is considered a verification failure. However, when the expected response time of M_Q is close to the time limit, small deviations in the response time can push some mutants into failure. This might give a false impression of instability, while in reality the solver behaves stably given enough time.

To address this issue, we further parameterize the categorization scheme with a tolerance factor ω between 0 and 1. When mixed results are observed in the samples $\hat{M}_Q$, we estimate the expected response time for M_Q using the mean response time of successful samples, denoted as $\hat{T}$. If the latter is close to the time limit, i.e., $\hat{T} \geq \omega T_{\text{limit}}$, the failures may be due to an insufficient T_{limit}. In that case, we take a conservative approach and do not label (Q, S) as `unstable`.

Figure 2 shows the tolerance test in the workflow. In our experiments, we use $\omega = 0.8$, and $T_{\text{limit}} = 60$s. Section IV gives a more detailed analysis of the impact of T_{limit}.

E. Results from Different Mutation Methods

The discussion about the workflow thus far has been based on a single mutation method. In our study, we consider **shuffling**, **renaming**, and **reseeding**, each of which outputs a stability category through our scheme. We use the following procedure to combine the results.

1) If the results are unanimously `inconclusive`, output `inconclusive`.
2) Remove `inconclusive` results. If the rest are unanimously X, output X.
3) Otherwise output `unstable`.

Note that if the mutation methods disagree on the categories, the procedure returns `unstable`. For example, if **shuffling** outputs `stable`, but **reseeding** outputs `unsolvable`, then the final result is `unstable`. In Section IV we show how mutation methods differ in their ability to detect instability.

F. Quantifying (In)stability

Given a query-solver pair (Q, S), we use the categorization scheme to answer the question of whether the pair is stable. To quantify the instability of an `unstable` pair, we simply use the **Mutant Success Rate** (from Section III-C) as a metric, where higher values are preferable.

To quantify the stability of `stable` queries, we use the **Standard Deviation of Mutant Response Times**. As discussed in Section I, increased response time impedes the iterative development cycles. Therefore, even if a query-solver pair is consistently producing the same verification result, a large variation in response time is still undesirable to the developer. Moreover, such variation is indicative of potential instability: if the time limit is shortened by a small amount, some mutants may fail to finish in time. Therefore, the larger the standard deviation, the less stable (Q, S) actually is.

IV. Experiments

We have presented a general methodology to detect and quantify SMT-based proof instability. To better understand the status quo of instability, we implement our methodology in the Mariposa tool and use it to perform experiments on existing program verification projects.

In this section, we first describe the experimental setup, which includes an overview of the Mariposa tool (Section IV-A), the verification projects studied (Section IV-B), and the configurations used (Section IV-C). We then present the experimental results, which are organized as a series of research questions (Section IV-D).

A. The Mariposa Tool

We implement our methodology in Mariposa, a tool for SMT stability testing. In its basic use case, Mariposa inputs a query-solver pair (Q, S), performs mutations on Q, runs S on the mutants, analyzes the performance data, and outputs the stability category and metrics.

For efficient manipulation of queries, the mutations are implemented using Rust ($\sim$200 LoC). The scripts for running the mutants, recording performance, and analyzing data are implemented in Python ($\sim$2K LoC).

Mariposa is extensible, so new mutation methods can be easily added. Mariposa is also configurable, allowing the user to specify parameters such as the number of mutants, the time limit, etc.

B. Projects Under Study

We experiment with prior automated program verification projects. For verification tools, we mainly focus on F* [20] and Dafny [19], since (1) they have been used to develop complex verified systems; (2) each has an active community of users; (3) they are actively maintained. We then select the following projects and extract all of the SMT verification queries they generate.

- Komodo$_D$. Komodo [24] is a security hypervisor verified and implemented in Dafny, a general-purpose program verifier that often generates undecidable queries.
- Komodo$_S$. Another research team reimplemented parts of Komodo using the Serval framework [2], which requires developers to work within a decidable fragment of first-order logic. For example, recursive functions and loops must be statically bounded. The goal is for developers to write fewer proofs, but one might also conjecture that using a simpler logic would lead to greater query stability.
- VeriBetrKV$_D$. VeriBetrKV [3] is a key-value store based on a B$^\varepsilon$ tree [36], implemented and verified in Dafny. VeriBetrKV$_D$ uses Dafny's standard dynamic frames [37] for heap reasoning.
- VeriBetrKV$_L$. In a follow-up study [38], researchers modified the VeriBetrKV code base to use a customized Dafny version that employs linear types for heap reasoning. They found that using linear types results in faster queries. We explore whether linear types also result in more stable queries.
- DICE$^\star_F$. DICE is an industry standard measured boot protocol [39]. DICE* [40] is a provably-correct implementation of the protocol in F*.
- vWasm$_F$. WebAssembly (Wasm) is a portable binary instruction format for web applications [41]. vWasm [42] is a provably-safe sandboxing compiler from Wasm to native code, implemented in F*.

Project	Source LoC	Query Count
Komodo$_D$	26K	2,054
Komodo$_S$	4K	773
VeriBetrKV$_D$	44K	5,325
VeriBetrKV$_L$	49K	5,600
DICE$^\star_F$	25K	1,536
vWasm$_F$	15K	1,755

TABLE I
Basic Statistics on Projects Used in Our Experiments

These project all exhibit non-trivial complexity. The source lines of code (LoC) and query counts for each project are summarized in Table I.

C. Experiment Configurations

We run the 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. Recapitulating earlier parameter settings, we set $T_{\text{limit}} = 60$s; 60 samples per mutation method; an alpha level of 0.05; $\omega = 0.8$; $r_{\text{solvable}} = 5\%$; and $r_{\text{stable}} = 95\%$.

For our experiments, we focus on the Z3 SMT solver [14], which all of our experiment projects were developed with, except for Komodo$_S$, which used both Z3 and CVC4 [43][3]. We are interested in both the current and historical status of SMT stability. Therefore, in addition to the latest Z3 solver (version 4.12.1, as of this writing), we include seven legacy versions of Z3, with the earliest released in 2015. In particular, for each project we include its *artifact solver*, which is the version used in the project's official artifact.

D. Experimental Results

We organize our experimental results around a series of research questions (RQs). Where necessary for space, we present the results from a subset of projects here and defer the rest to a technical report [44].

[3]We had initially planned to run our experiments with cvc5 [15] too. However, our preliminary experiments showed the projects are overfitted to Z3. Without intervention, cvc5 cannot solve any of the Dafny or F* queries, since it cannot even parse the SMT queries these program verification tools produce, due to their use of various bits of Z3-specific syntax and features. After we converted the queries to a format cvc5 understands, it could only solve $\sim$14% of the queries in Komodo$_D$. We consulted with the cvc5 developers for option tuning and tried cvc5's automated configuration script for SMT-COMP, but it did not significantly improve the number of queries solved.

RQ1: Do Solver Upgrades Improve Stability?

For each query-solver pair (Q, S), we run Mariposa, which outputs a stability category. In Figure 3, each stacked bar shows the proportions of categories in a project-solver pair. In all project-solver pairs, the majority of queries are stable. However, a non-trivial amount of instability persists as well.

We observe different trends in each project as newer solver versions are used. The `unstable` proportion of vWasm$_F$ and Komodo$_S$ remain consistently small across the tested solver versions. On the other hand, we observe signs of projects that "overfit" to their artifact solver, in that they become less stable with solver upgrades.

Specifically, all of the Dafny-based projects in our study show more instability in newer Z3 versions, with a noticeable gap between Z3 4.8.5 and Z3 4.8.8. The difference in the stability performance is perhaps expected, as these projects were all developed using (now) outdated Z3 solver versions. As of the time of writing, F* continues to use Z3 4.8.5, which is approximately four years old, while Dafny only transitioned away from that version earlier this year.

Commit Bisection. We perform further experiments to narrow down the Z3 git commits that may have caused the increase in instability. In the six experiment projects, 285 queries are `stable` under Z3 4.8.5 but `unstable` under Z3 4.8.8. For each query in this set, we run `git bisect` (which calls Mariposa) to find the commit to blame, i.e., where the query first becomes `unstable`.

Table II shows the the bisection results for the 285 queries. Note `git bisect` might not be able to find a unique commit to blame. For example, when the binary search narrows the problem down to a region where commits do not compile, all commits in that region are potentially to blame. We indicate such cases as N/A in the table.

There are a total of 1,453 commits between the two versions, among which we identify two commits that have the most impact. Out of the 285 queries, 115 (40%) are blamed on commit `5177cc4`. Another 77 (27%) of the queries are blamed on `1e770af`. The remaining queries are dispersed across the other commits.

These two most significant commits are small and localized: `5177cc4` has 2 changed files with 8 additions and 2 deletions; `1e770af` has only 1 changed file with 18 additions and 3 deletions. Both commits are related to the order of flattened disjunctions. `1e770af`, the earlier of the two, sorts the disjunctions, while `5177cc4` adds a new term ordering for ASTs, which it uses to replace the previous sorting order of disjunctions.[4]

[4]We contacted the Z3 developers after our paper submission. Coincidentally, they were investigating regressions in F* query success, and they identified the same two commits as having the most significant impact. Their fix is now merged into the Z3 master branch.

hash	blames	commit message
`5177cc4`	115	`change lt`
`1e770af`	77	`local sort`
`db87f2a`	16	`separate rewriter...`
`ff6b330`	12	`remove incorrect ...`
`7f073a0`	7	`fix #2452 fix #...`
`8b23a17`	3	`move flatten func...`
`c70e9af`	3	`fix #3734`
`dd452e0`	3	`eq`
`762f265`	3	`merge with master`
`001ddef`	3	`fix #2749`
`3774d6d`	2	`fix #2890`
`3ef05ce`	2	`tuning`
`80994f7`	1	`redirect to the n...`
`d23230e`	1	`fix declaration s...`
`e5dffea`	1	`fix #2365`
`ad55a1f`	1	`Update release.ym...`
`06ee09a`	1	`Update README.md`
`38ad66c`	1	`update hash #257...`
`9cccfb9`	1	`Take one on addin...`
`ba40a57`	1	`better branching ...`
`1e92165`	1	`branch selection ...`
`bba2cf9`	1	`fix #3163`
`2a1f8ac`	1	`revert normalizin...`
N/A	28	
total	285	

TABLE II
COMMIT BISECTION RESULTS

RQ2: Do Projects Differ in Stability?

Komodo$_D$ and Komodo$_S$ are two implementations of the Komodo security hypervisor in Dafny and Serval respectively. The `unstable` proportion of both projects is small using their artifact solvers. However, Komodo$_D$ shows a significant increase in instability using newer versions of Z3, while Komodo$_S$ remains stable. Note that Komodo$_S$ implements a subset of the features in Komodo$_D$. If we exclude the attestation-related queries from Komodo$_D$, which are not present in Komodo$_S$, the `unstable` proportion of Komodo$_D$ is reduced to 4.27% (from 5.01%) using the latest Z3. The proportion is still much higher than Komodo$_S$'s (0.52%). The gap may be attributable to other differences in features and proof goals, but it may also indicate that restricting queries to a *decidable* logic (as Komodo$_S$ does) improves stability.

VeriBetrKV$_L$ and VeriBetrKV$_D$ are two implementations of VeriBetrKV in Dafny with different approaches to heap reasoning, where the authors of VeriBetrKV$_L$ report better query times by adopting linear types. However, their result does not appear to generalize to stability performance: VeriBetrKV$_L$ is only slightly more stable than VeriBetrKV$_D$ when using their artifact solvers, and both suffer similar stability regressions on later solvers.

We notice that vWasm$_F$ is remarkably stable: the `unstable` proportion of vWasm$_F$ is almost negligible across all solver versions. We contacted the authors of vWasm$_F$, and they confirmed that they put significant

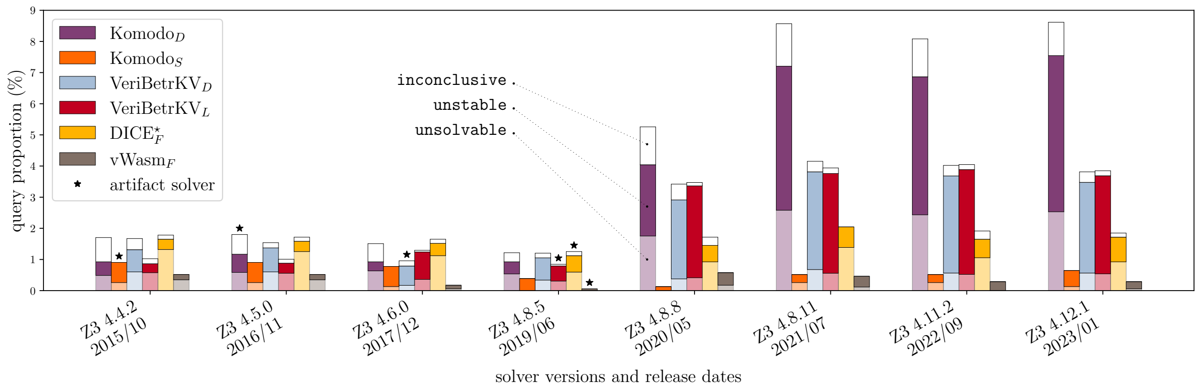

Fig. 3. **Overall Stability Status.** From bottom to top, each stacked bar shows the proportions of `unsolvable` (lightly shaded), `unstable` (deeply shaded), and `inconclusive` (uncolored) queries. The remaining portion of the queries (stacking each bar to 100%), not shown, are `stable`. The solver version used for each project's artifact is marked with a star ($\star$).

manual engineering effort into making the queries stable [45]. They attribute the stability of their queries to a disciplined usage of multiple empirically developed techniques. Globally, they disable the non-linear arithmetic solver (anecdotally prone to instability), reduce F*'s `fuel/ifuel` settings (which control unrolling of recursive functions and inductive data types), and minimize the use of ambient lemmas (that tend to bloat solver context). They also minimize the use of (user-introduced, F*-level) quantifiers, and manually pick good quantifier triggers. Particularly complex proofs necessitated even more drastic measures: using F*'s tactic framework to perform manually-controlled normalization of terms before verification condition generation. They note that neither the original un-normalized nor the fully-normalized forms were amenable to stable proofs; only the manually controlled normalization worked.

While few projects can afford this level of manual effort, these results suggest that developers and program verification frameworks can potentially shape their queries to minimize instability.

RQ3: Do Longer Time Limits Mitigate Instability?

As we discussed in Section III-D, the choice of time limit T_{limit} could impact our experimental results. Indeed, one might expect that `unstable` queries will eventually turn into `stable` ones given large enough time limits. To test this hypothesis, we extended the experiments using the most recent Z3 (version 4.12.1) with a time limit of 150s (2.5 $\times$ 60s).

In Figure 4 we report the proportion of `unsolvable` and `unstable` queries for each T_{limit} in Komodo$_D$ and VeriBetrKV$_D$. We observe that the `unsolvable` proportion drops as T_{limit} increases. This is expected, as a query might only become solvable with a longer time.

However, the `unstable` proportion stays remarkably consistent after initial fluctuations. That is, certain `unstable` queries *remain* `unstable`, even with a longer time limit. To analyze this further, we report the intersection of `unstable` queries at T_{limit} and $T_{\text{limit}}+\text{step}$, for steps of 10, 30 and 60 seconds. One can interpret a $T_{\text{limit}}+\text{step}$ curve as follows: if some queries are `unstable` at T_{limit}, it reports how many of them will remain `unstable` at $T_{\text{limit}}+\text{step}$.

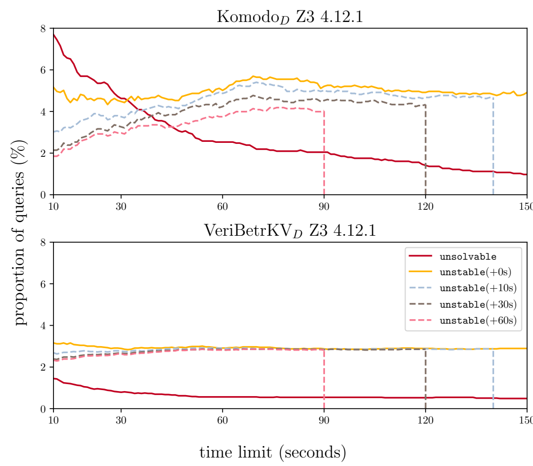

Fig. 4. **Comparing Time Limit Choices.** The proportion of `unstable` queries stays around the same level after some fluctuations.

We observe that for a step of 10s, the difference is small. This means that most `unstable` queries remain `unstable` if given 10 more seconds, which is expected. For a step size of 60s, the difference is larger but still not significant. In VeriBetrKV$_D$, it has almost no impact beyond 30s. Therefore, while a longer time limit could help mitigate instability, it is not a silver bullet.

RQ4: Do Results from Mutation Methods Overlap?

We covered multiple mutation methods in our study. A natural question is whether these methods are equally effective in detecting instability.

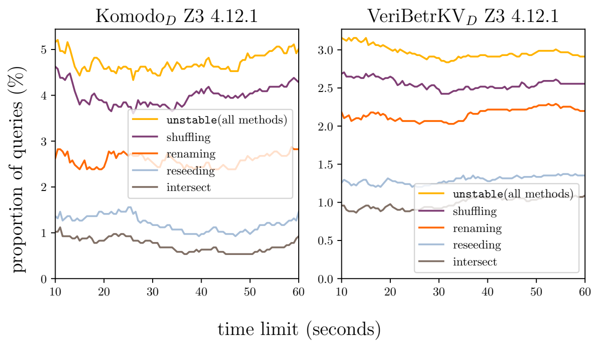

Fig. 5. **Comparing Mutation Methods**. Mutation methods differ in their ability to detect instability. Z3 4.12.1 has the most instability found through **shuffling**. The intersection of methods is also shown as a reference. Note that each sub-graph uses a different y-axis scale.

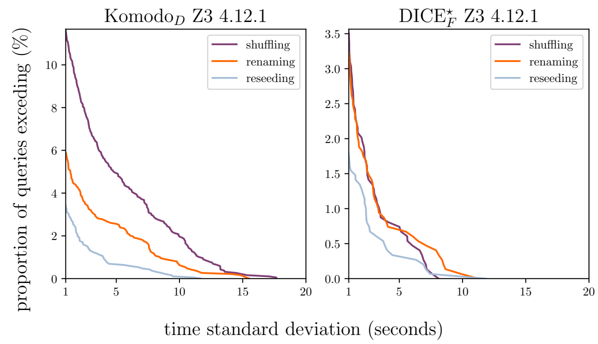

Fig. 6. **Degree of Stability in Stable Queries**. The proportions are taken over `stable` queries only, some of which still exhibit large standard deviations in time among mutants. Mutation methods also differ in their impact. Each sub-graph uses a different y-axis scale.

In Figure 5, we show the unstable proportions identified using each mutation method, along with the overall `unstable` proportion. Recall that the latter is a superset of the individual mutations, as discussed in Section III-E. Since the choice of T_{limit} may impact the categorization, we present results for different T_{limit} as well.

Our results indicate that the effectiveness of mutation methods differ. For example, in Komodo$_D$ and VeriBetrKV$_D$, the `unstable` proportion is the highest for **shuffling**, followed by **renaming**, then **reseeding**, regardless of T_{limit}. In fact, of the `unstable` queries in Komodo$_D$ at 60s, 36.9% are uniquely identified by **shuffling**, 6.8% by **renaming**, and 3.9% by **reseeding**.

RQ5: How Stable are Stable Queries?

The **Standard Deviation of Mutant Response Times** is a metric we introduced in Section III-F, where a large value indicates less actual stability, even if mutants consistently succeed. Figure 6 shows the distribution of standard deviation from `stable` queries, which are mostly less than 1s, but there are exceptions exceeding 10s, which is significant given the 60s limit.

RQ6: Is the Original Query Special?

In our methodology, the original query is treated as a member of the mutant set. It might be reasonable to ask how does the original query differ from its mutants in terms of performance.

In Figure 7, we show the verification time of the original query and the median of its mutants, using the data from our extended time limit experiment. In Komodo$_D$, which has the highest `unstable` proportion among the six projects, the run time of the original and its mutants are generally within ±50% of each other. In vWasm$_F$, where the `unstable` proportion is the lowest, the two have nearly identical performance.

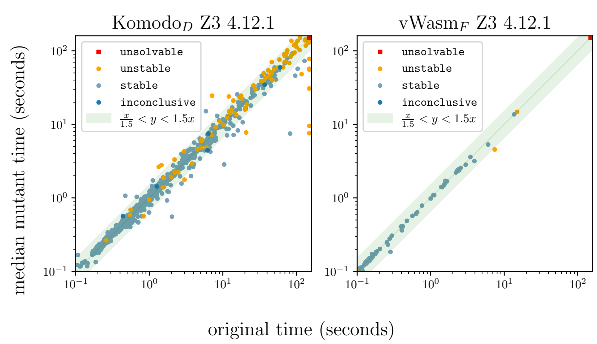

Fig. 7. **Comparing Original and Mutant Queries.** The original query has a run time similar to the median of the mutants. In Komodo$_D$, a few cases have the original query time out while some mutants succeed.

V. The Mariposa Benchmark

Our experiments over a total of 17,043 original queries generate more than 3 million mutants and take more than 578 CPU days to evaluate. To facilitate future research, we distill the experiment queries into the Mariposa benchmark set. We hope this first version of Mariposa will incentivize improvement and prevent regressions in SMT solver stability for program verification workloads.

The Mariposa benchmark includes both unstable and stable queries for the projects we experimented with, as shown in Table III. Each is further divided into a core and an extension, where the core contains fewer but more representative queries.

The unstable core set contains the queries from each project that are categorized as `unstable` in both the artifact solver and the latest solver. These queries have been consistently unstable, which might be indicative of a long-term problem. The extension set contains all the additional `unstable` queries in the latest Z3 version.

Project	Original	Unstable core	Unstable ext.	Stable core	Stable ext.
Komodo_D	2,054	8	95	30	97
Komodo_S	773	2	2	30	9
VeriBetrKV_L	5,600	22	153	30	102
VeriBetrKV_D	5,325	25	130	30	81
$\text{DICE}^{\star}_F$	1,536	3	9	30	13
vWasm_F	1,755	0	4	30	0
Total	17,043	60	393	180	302

TABLE III
THE MARIPOSA BENCHMARK

The stable core set contains 30 randomly selected `stable` queries from each project, with mutant time standard deviation less than one second. This set is meant to prevent stability regression, since each member has a consistent result and run time. The extension set contains all the `stable` queries that have a standard deviation of more than six seconds. Given that the time limit is 60 seconds, such large standard deviations may indicate potential instability, as discussed in Section III-F.

VI. LIMITATIONS

Our experiments draw from six verification projects, which we cannot claim are fully representative of all the SMT-based program verification projects. Nevertheless, we believe our experiments offer valuable insights and serve as a starting point for future work.

Our experiments are performed only with Z3. As explained earlier, popular automated verification languages such as Dafny and F* emit queries that are overfitted to Z3. Hence, our results may not extend to other solvers, such as cvc5.

Our mutation methods are not exhaustive. This study explores a few common mutations, but there are many other mutation methods that might be of interest. For example, mixing mutations may expose more instability, e.g., performing **shuffling** and **renaming** at the same time. We leave the exploration of additional mutation methods to future work.

Our results are dependent on our choice of configuration parameters, e.g., the time limit, the alpha level, etc. In our experiments and analysis, we have tried to analyze the impact of these choices (e.g., via our additional experiments with extended time limits). However we cannot guarantee that our results are not sensitive to our particular choice of configurations.

Our results likely under represent actual instability in the development process. We note that the projects we studied are the cleaned up final versions of the code. During development, although developers do not typically test for instability, they usually try to fix the most obnoxiously unstable proofs.

VII. CONCLUSION

In this work we have studied the phenomenon of SMT instability, specifically in program verification projects, where changes are expected, responsiveness is preferred, and stability is critical. We have proposed a new methodology for detecting and measuring instability, which can inform program verification tools of instability in generated queries. We have also constructed a new benchmark suite, which can be used by SMT solvers to evaluate and optimize for stability. We have applied our methodology to evaluate a number of existing verification projects on various solver versions. Our results show that:

1) Stability is the common case, but instability exists non-trivially: 2.6% of the queries in our experiment are unstable with the most recent Z3. In specific projects, this ratio can be as high as 5.0%.
2) Stability may deteriorate with solver upgrades: three out of six projects in our experiment show notably worse stability on newer solver versions.
3) Mutation methods differ in their effectiveness in detecting instability. Specifically, currently employed detection methods based on random seeds only capture a fraction of the problem.
4) Mutants of a given query can exhibit large run-time variance, even if consistent in verification results.
5) Source-level program changes may reduce instability, but this currently requires extensive manual engineering. For example, limiting the use of quantifiers, nonlinear arithmetic, or undecidable theories may help.
6) Increasing the time limit for queries can improve stability, but it offers diminishing returns.

ACKNOWLEDGMENTS

Chris Hawblitzel and Doug Woos contributed to an initial exploration of SMT instability in 2016. Mariposa is a complete rewrite and a fresh set of experiments. We thank Andrew Reynolds for his help with cvc5 configuration; Jinjin Tian for her advice on our statistical analysis; Guido Martinez and Nikolaj Bjørner for sharing their work on patching Z3; and Joshua Gancher, Nikhil Swamy, and the anonymous reviewers for their helpful feedback on the paper.

This work was supported in part by the National Science Foundation (NSF) under grants 1901136, 2015445, and 2224279, grants from the Intel Corporation and Rolls-Royce, Amazon Research Awards (Fall 2022 CFP), the Prabhu and Poonam Goel Graduate Fellowship, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of these supporters.

REFERENCES

[1] 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.

[2] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[3] 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.

[4] 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.

[5] Y.-F. Fu, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Signed Cryptographic Program Verification with Typed CryptoLine," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019. [Online]. Available: https://doi.org/10.1145/3319535.3354199

[6] 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

[7] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical Verification for Programmable Data Planes," in *Proceedings of ACM SIGCOMM*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 490–503.

[8] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety Verification of Smart Contracts," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.

[9] 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 *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[10] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran *et al.*, "Continuous Formal Verification of Amazon s2n," in *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2018.

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

[12] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Springer, 2018.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] "Debugging Unstable Verification,"

http://dafny.org/dafny/DafnyRef/DafnyRef.html#1365-debugging-unstable-verification.

[22] "Repeating Proofs with Quake," http://www.fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html#repeating-proofs-with-quake.

[23] C. Barrett, L. de Moura, and A. Stump, "SMT-COMP: Satisfiability modulo theories competition," in *Computer Aided Verification*, 2005.

[24] 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.

[25] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "SAT Competition 2020," *Artificial Intelligence*, vol. 301, 2021.

[26] T. Weber, "Scrambling and Descrambling SMT-LIB Benchmarks," in *SMT @ IJCAR*, 2016.

[27] A. Biere and M. Heule, "The Effect of Scrambling CNFs," in *Proceedings of Pragmatics of SAT*, vol. 59, 2019.

[28] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "StringFuzz: A Fuzzer for String Solvers," in *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2018.

[29] D. Winterer, C. Zhang, and Z. Su, "On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers," vol. 4, no. OOPSLA. Association for Computing Machinery, Nov. 2020.

[30] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative Type-Aware Mutation for Testing SMT Solvers," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485529

[31] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Skeletal Approximation Enumeration for SMT Solver Testing," in *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2021.

[32] A. Niemetz, M. Preiner, and C. Barrett, "Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers," in *Computer Aided Verification*, 2022.

[33] A. Bugariu and P. Müller, "Automatically Testing String Solvers," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1459–1470.

[34] M. Bringolf, D. Winterer, and Z. Su, "Finding and Understanding Incompleteness Bugs in SMT Solvers," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560435

[35] W. Feller, *An Introduction to Probability Theory and its Applications, Volume 2*. John Wiley & Sons, 1991, vol. 81.

[36] G. S. Brodal and R. Fagerberg, "Lower Bounds for External Memory Dictionaries," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[37] I. T. Kassios, "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions," in *Proceedings on the International Symposium on Formal Methods (FM)*, 2006.

[38] 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.

[39] A. Marochko, D. Mattoon, P. England, R. Aigner, R. Spiger (CELA), and S. Thom, "Cyber-Resilient Platforms Overview," Microsoft Research, Tech. Rep. MSR-TR-2017-40, September 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cyber-resilient-platforms-overview/

[40] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur, "DICE*: A Formally Verified Implementation of DICE Measured Boot," in *Proceedings of the USENIX Security Symposium*, Aug. 2021.

[41] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

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

[43] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011.

[44] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, "Mariposa: Measuring SMT Instability in Automated Program Verification (Technical Report)," Carnegie Mellon University, Tech. Rep., August 2023. [Online]. Available: https://doi.org/10.1184/R1/23905905

[45] J. Bosamiya, W. S. Lim, and B. Parno, private communication, 2023.