Towards Scalable Automated Program Verification for System Software

Yi Zhou

CMU-CS-25-101

May 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Bryan Parno (Chair)
Marijn Heule
Ruben Martins
Jon Howell (University of Washington)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

The work presented in this thesis was supported in part by National Science Foundation under grants 2224279, 2318953, 1901136, 2015445, 1801369, DGE-1745016, DGE-1762114, National Science Foundation/VMware under grant CNS-1700521, Department of the Navy/Office of Naval Research under grant N00014-17-S-B001, Air Force Research Laboratory/Defense Advanced Research Projects Agency under agreement FA8750-24-9-1000, Alfred P. Sloan Foundation, Amazon Research Award, Carnegie Mellon CyLab Future Enterprise Security initiative, Google Faculty Fellowship, Google PhD Fellowship, Intel Corporation, Kwanjeong Educational Foundation, Prabhu and Poonam Goel Graduate Fellowship, Rolls-Royce, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these sponsors.

Keywords: Program Verification, System Software, First-Order Logic, Satisfiability Modulo Theories

To Sisyphus.

Abstract

Automated Program Verification (APV) provides formal guarantees about software while promising strong automation in the verification process. APV has already seen preliminary successes in system software (e.g., file systems, network protocols), extending beyond academic prototypes to industrial applications. However, the scalability of APV becomes an issue as we move towards more complex systems, where automation failures start to show up. Such failures often require tedious manual fixes, breaking the pledge of automation in APV. Worse yet, since program verification is fundamentally undecidable, automation failures are inherently inevitable.

Nevertheless, that does not mean APV is hopeless beyond small-scale systems. In this thesis, we organize the discussion around the development stages of APV: (1) creating proofs, (2) reusing proofs, (3) debugging proofs, and (4) stabilizing proofs. We argue that, despite the undecidable nature of program verification in theory, we can overcome the scalability challenges that arise in practice, due to the recurrent patterns in APV programming and reasoning.

Specifically, we make empirical observations on the common motifs in APV, and then design formal methods to leverage them for automation. Using large-scale verified systems as case studies, we show this combination of formal and empirical methods leads to practical improvements in APV for system software.

Acknowledgments

I would like to thank my advisor, Bryan Parno, for his guidance and encouragement over the past six years. The significant freedom he grants to his students, the remarkable understanding he has on the projects, the impressive quality of the research he leads, and the meticulous attention he pays to exhibition have been invaluable to my growth as a researcher.

I would also like to thank the rest of the committee members, Marijn Heule, Ruben Martins, and Jon Howell, for their insightful feedback on the thesis. Marijn Heule has provided crucial guidance on my line of work in stability, offering deep insights into the domain of SAT/SMT solvers. Jon Howell has profoundly influenced my work with his intuitive explanations of formalism and his developer-focused perspectives.

I would like to express my gratitude to additional individuals who have played pivotal mentorship roles in my journey into formal methods. I would like to thank Jay Bosamiya, who has directly or indirectly influenced almost all of my doctoral work. I would like to thank Travis Hance, who has also served as a role model as a senior student. I would like to thank my manager Byron Cook during my internship at AWS. His perspective on the industry has been invaluable to my understanding of the real-world applications of formal methods.

I would like to thank my other collaborators and co-authors that made this thesis possible, including but not limited to Andrea Lattuada, Amar Shah, Chanhee Cho, Chris Hawblitzel, Felix Miller, Jade Philipoom, Jessica Li, Jialin Li, Jonathan Cameron, Jonathan Protzenko, Menucha Winchell, Rob Johnson, Sydney Gibson, Sarah Cai, Yoshiki Takashima, and Zhengyao Lin.

As for the empirical side of my work, I owe a lot to to my former advisor, Michael Bailey, for his guidance during my undergraduate and master's studies. I have learned a lot about empirical methods from Bailey, who has made a long-lasting impact on my research career. The time in NSRG also greatly changed my personality. I would like to thank Joshua Mason, Deepak Kumar, Zane Ma, and Simon Kim, who have been great mentors and friends.

I would also like to thank the staff and administrators at CMU and UIUC, including but not limited to Brittany Frost, Deborah Cavlovich, Matthew Stewart, Maverik Woo, and Steve Herzog. Their administrative and/or technical support has been indispensable to my study and research.

Aside from research, health is a crucial part of life, if not the most important one. I would like to express my gratitude to my psychiatrist, Rebekah DeChicko, for her help in my mental health over the years. I would also like to thank my orthopedists, Xiaoxing Jiang and Zixian Chen for their excellent surgeries.

Last but not least, I would like to thank my family. In the past decade, I have been following the footsteps of my cousin, Hao Zhang, who has helped me a lot in my life in the US. I would like to thank my parents, Shuying Liu and Guanghua Zhou, who have always been supportive of my decisions. Their

dedication to medicine, combined with their genuine compassion for patients and their remarkable composure in the face of life and death, has been a source of inspiration for me. Since they named me after the ancient Chinese Book of Changes, I shall preface this thesis with a quote from the book:

天行健 君子以自強不息
地勢坤 君子以厚德載物

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Scalability Problems

As programmers, we often make various informal claims about our software, including its correctness, efficiency, security, and so on. However, as many of us can also testify, these claims are sometimes unsubstantiated or even untrue. Luckily, *formal verification* offers a path to move away from informal claims and towards formal guarantees about programs.

Formal verification uses mechanized proofs to show that the code meets its specification, precluding entire classes of problems (e.g., buffer overflows and race conditions) that have plagued traditional software development. In particular, *Automated Program Verification* (APV) offers an encouraging approach, with a promise of strong *proof automation*. APV features the use of Satisfiability Modulo Theories (SMT) solvers [15]. Specifically, APV languages encode the proof obligations as SMT queries, where state-of-the-art SMT solvers such as Z3 [47] and cvc5 [10] can often discharge the obligations automatically, eliminating a significant number of manual proof steps.

APV has received increasing interest in recent years. This is evidenced by a growing list of APV languages, which includes Dafny [97] and F* [154] with active support from Amazon Web Services and Microsoft, respectively. Moreover, APV has seen preliminary success in various *system software* domains, including distributed systems [78, 104, 110], storage systems [6, 24, 72, 100], operating systems [26, 59, 74, 79, 123, 124, 149, 155], networking systems [9, 20, 37, 39, 48, 49, 80, 101, 167, 171], and security/cryptography [2, 3, 23, 137, 139, 140, 174, 177].

However, when we apply existing APV techniques to larger systems, *scalability* becomes an issue. There can be many concrete symptoms, but they all boil down to failures in proof automation. Fixing these failures often requires tedious efforts and advanced expertise. Put in a crude way: automation failures can be laboriously fixed by Ph.D. students, but that is not a practical strategy for production purposes.

Worse yet, the scalability challenges are fundamentally due to *undecidability*. As Rice's theorem [144] underscores, verifying programs with basic language features such as recursion is generally undecidable. Therefore, APV techniques suffer from inherent *incompleteness*. Informally, no matter how powerful the SMT solvers or APV languages become, there will always be "simple" programs that cannot be automatically verified.

1.2 The Recurrent Patterns

Even though the theoretical limitation seems daunting, all hope is not lost yet. In this thesis, we discuss a few major challenges to the scalability of APV for system software. We highlight the *recurrent patterns* in programs or proofs of interest, and then exploit these patterns to address various scalability problems. Specifically, our thesis statement is:

> *While fully automated program verification is impossible,*
> *we can often have scalable solutions to practical systems,*
> *based on the recurrent reasoning and programming patterns.*

In this thesis, we demonstrate the principle throughout the lifecycle of APV projects, from creating and debugging proofs in active development, to reusing and stabilizing proofs for long-term maintenance. We employ *empiricism* when applicable, to help discover the recurrent patterns and evaluate our formal techniques. We show that the combination of formal and empirical methods can lead to pragmatic improvements in the scalability of APV.

1.2.1 Creating Proofs

We begin the main discussion in Chapter 3, in which we cover proofs about memory safety and theory-specific reasoning. We place an emphasis on the former, since memory safety is a prerequisite for more advanced properties such as security and correctness.

We first illustrate the challenges of memory safety proofs in Sec. 3.1.1, using QUIC$_D$ [49] as a case study. QUIC$_D$ is our implementation of the QUIC [93] protocol in Dafny (~10kLoc). As we build up the data structure hierarchy in our implementation, the memory invariants become increasingly complex. Consequently, the SMT solver often struggles to prove the safety of heap updates, especially for the higher-level structures. This challenge arises because Dafny's memory model allows arbitrary aliasing – the memory invariants must account for possible points-to relation between each pair of structures. Meanwhile, we also observe that over 90% of the memory updates in our implementation do not actually involve aliasing.

In Sec. 3.1.3, we build on this observation and propose a novel approach to memory safety proofs. Specifically, we introduce linear types to Dafny [100], creating a hybrid of linear and regular regions in the memory model. Specifically, each linearly-typed structure

enforces a unique owner, eliminating the need for further aliasing reasoning at the SMT level. Meanwhile, we retain the flexibility of arbitrary aliasing in regular regions whenever needed. We demonstrate the effectiveness of this approach, by reproducing an implementation of the VeriBetrKV key-value store (~24 kLoc) using 28% fewer lines of proofs and 30% faster verification time than its vanilla counterpart. The result further inspires us to design a new APV language, Verus [94], tailored for the Rust programming language that already features an affine type system.

Beyond memory safety, specialized theories are another building block in APV. In particular, cryptography and systems programming often involve bit-level operations, which requires reasoning over nonlinear integer arithmetic (NIA) and bit-vectors (BV). In Sec. 3.2.1 and Sec. 3.2.2 we demonstrate the complications these theories introduce and the manual workarounds in Dafny programs. We then then describe how we encode these proof obligations in Verus, automating the manual proof patterns.

1.2.2 Debugging Proofs

Despite the best efforts to automate proofs, verification failures are inevitable. In Chapter 4, we discuss the challenges in debugging proofs. Current APV languages provide limited insight into the cause of failures. In practice, developers often resort to a manual trial-and-error process. Similar to debugging traditional software using print statements, the process can be ad-hoc and time-consuming, which slows down development. Nonetheless, over the years, we have observed recurrent patterns in the manual debugging practice. We thus create the ProofPlumber framework [36], which provides an API for developers to easily define their own debugging strategies for Verus proofs. We call such debug automation *proof actions*, which is analogous to the concept of *code actions* in modern IDEs. As a proof of concept, we implement 17 common proof actions with ProofPlumber API, using only 29–177 lines of code each.

1.2.3 Reusing Proofs

We then transition our discussion from active development to long-term maintenance in APV. In Chapter 5, we focus on reusing proofs, which is similar to reusing code in traditional software development. More specifically, we discuss proof reuse in the context of verifying low-level cryptography across multiple heterogeneous hardware platforms through our Galápagos framework [174]. We demonstrate that, with the introduction of verified functors for proof reuse, it is possible to achieve a 28%–65% reduction in verification effort when supporting additional platforms or algorithms. As part of the Galápagos framework, we have also curated the first Dafny standard library (~5.8 kLoc), which generalizes and unifies the previously standalone libraries spread across past projects. The library is now a part of the Dafny distribution [43], providing reusable lemmas for the community and also attesting the recurrent reasoning patterns in APV.

1.2.4 Stabilizing Proofs

While the reusing/debugging proofs has an intuitive counterpart in traditional software development, in Chapter 6 we discuss a unique challenge in APV, *stability*. Specifically, *proof instability* (also called brittleness) refers to the phenomenon where non-semantic changes to the program create spurious verification failures. Fixing such failures incurs significant maintenance overhead without providing new insights into the program under verification. Worse yet, the problem tends to be exacerbated in large-scale projects with multiple collaborators and frequent code changes, creating a major scalability challenge.

Our first step towards addressing instability is to quantify it. In Sec. 6.1, we introduce Mariposa [173], a tool to detect and measure instability. Mariposa takes as input an SMT query-solver pair, mutates the query with non-semantic changes, and outputs a stability judgment. Intuitively, if the solver's performance varies among the mutated queries in a statistically significant manner, Mariposa flags the query-solver pair as unstable.

We then use Mariposa to conduct a large-scale empirical study, analyzing over 26,000 queries collected from 14 existing system verification projects written in Dafny, F*, Verus, and Serval [122]. In some projects, the ratio of unstable queries can be as high as 5%. While this might not seem significant at first glance, for a regular software project, it would be completely unacceptable to have 5% of its unit tests fail randomly. Unfortunately, for APV developers, instability has been a persistent burden to bear. To highlight this issue, we curate the first APV query benchmark, which includes a unstable set to test for improvements and a stable set to watch for regressions.

As we have a way to quantify instability and a benchmark suite to evaluate against, we start working on mitigation, using a combination of empirical and formal methods. In Sec. 6.2, we present a novel SMT pre-processing algorithm named Shake [172]. We base Shake on the observation that APV queries exhibit a *goal-axiom* structure, which distinguishes them from general SMT queries. In this structure, the goal represents the intended property to be verified, while the axioms supply the background information. Through idealized experiments, we discover that 96%–99% of the axioms are irrelevant to the goal, while accounting for a vast majority (78%) of the instability instances. Motivated by the finding, we design Shake to prune out less relevant axioms from the queries, which mitigates instability by 29% on Z3 and 41% on cvc5.

In Sec. 6.3, we further leverage the goal-axiom structure to repair instability at a fine-grained level. Through carefully controlled experiments, we discover that instability is often due to a few *query-specific* axioms. With this empirical insight, we design Cazamariposas, a tool to identify the problematic axioms along with fixes. Specifically, we present a novel differential analysis technique in Cazamariposas, taking advantage of the divergence in axiom usage profiles when a query undergoes Mariposa-style mutations. When we apply Cazamariposas to our benchmark suite, it repairs ~70% of the unstable queries, providing fixes involving ≤ 2 axioms.

1.3 The Concrete Contributions (TLDR)

This thesis addresses several major scalability challenges in Automated Program Verification (APV) for system software. More specifically, we discuss the challenges across the lifecycle of APV projects and propose pragmatic solutions based on recurrent patterns in programs or proofs of interest. Here we summarize the contributions in each chapter:

Creating Proofs.

- QUIC_D. A verified implementation of the QUIC protocol in Dafny.
- Linear Dafny. A language extension to Dafny, where we leverage linear types to simplify the reasoning over common-case memory-access patterns.
- VeriBetrKV_L. A verified implementation of the VeriBetrKV key-value store using Linear Dafny.
- Verus. An APV language for Rust programs, where we leverage the built-in affine type system to simplify memory reasoning and automate the manual proof patterns for NIA and BV.

Debugging Proofs.

- ProofPlumber. A framework for debugging Verus proofs, with an API to customize debugging strategies.
- Proof Actions. 17 common debugging strategies implemented with the ProofPlumber API, automating the corresponding manual debug patterns.

Reusing Proofs.

- Galápagos[1]. A framework for low-level cryptography on heterogeneous platforms, where we introduce verified functors to generalize and deduplicate proof patterns.
- Dafny Standard Library. The first curated library of reusable lemmas for the Dafny community, where we capture the recurrent reasoning patterns in past projects.

Stabilizing Proofs.

- Mariposa[2]. The first tool to systematically detect and measure instability, and the first APV query benchmark suite.
- Shake. The first SMT-level algorithm to mitigate instability, where we leverage the pattern of goal-axiom structures in APV queries.
- Cazamariposas[3]. The first SMT-level tool to automatically repair unstable queries, leveraging the differential patterns in axiom usage.

[1]The Galápagos finch and tortoise species are famous for adapting their bodies to the different environments on each of the Galápagos Islands. In the same vein, the Galápagos framework adapts cryptographic algorithms to the specifics of each supported hardware model.

[2]The name comes from the Spanish word for butterfly, where instability is similar to the so-called butterfly effect in chaos theory.

[3]The name comes from the Spanish word butterfly net, since we use this tool to fix instability.

Chapter 2

Background

In this chapter, we provide an overview of the automated verification of system software. In Sec. 2.1, we introduce deductive program verification, outlining its historical context and logical foundations. In Sec. 2.2, we discuss system software, which we focus our verification efforts on. More specifically, we discuss what the typical requirements for system software are, and how verification can help meet the requirements. In Sec. 2.3, we expand on automated program verification (APV), which is the main topic of our thesis. In particular, we cover the components of the APV pipeline, and discuss the current challenges and limitations of APV for system software.

2.1 Program Verification Overview

In deductive program verification, we use mechanized proofs to formally establish software properties. Typically we provide the following as inputs to a program verifier:

(1) Source code, which is meant to be compiled/executed.
(2) Specifications, which are usually non-executable logical formulas.
(3) Proofs justifying that (1) meets (2).

The deductive program verifier then outputs whether the proof is valid.

It is *believed* that verification can offer a high degree of assurance. In comparison to software testing, which checks a program's behavior on a limited subset of possible inputs, verification ensures that a program complies with its specifications on all inputs. Please note that in this thesis we do not go to the lengths to justify the business or fiscal value of program verification as a development method. Instead, we work under the assumption that deductive verification is a sensible endeavor in academia and certain industry settings.

2.1.1 Abbreviated Histories of Verification

Deductive program verification originated from the work of Floyd and Hoare [61, 81] in the 1960s. Floyd-Hoare logic, often simply referred to as Hoare logic, is a formal logic based on *Hoare triples*, which are statements about program properties. A Hoare triple is of the form $\{P\}C\{Q\}$, where P and Q are logical predicates on program state and C is a program. Intuitively, this triple states that if the program C is executed in a state satisfying the pre-condition P, it will terminate [1] in a state satisfying the post-condition Q.

Hoare logic also includes mechanisms to prove these statements of program properties. Given the syntax and semantics of a language L, we can mechanically derive its Hoare logic, which would provide a set of inference rules capturing the behavior of each basic language construct (e.g., assignment, sequencing, conditionals). We can thus build the proof of a Hoare triple $\{P\}C\{Q\}$ from the proofs of P's sub-constructs, which eventually boils down to the inference-rule applications.

Dijkstra's weakest pre-condition calculus [51] further simplifies the proof construction. Specifically, proving $\{P\}C\{Q\}$ is reduced to proving $P \Rightarrow wp(C, Q)$, where $wp(C, Q)$ is the weakest pre-condition the calculus computes. Furthermore, if the state predicates (i.e., P, Q) are within First-order logic (FOL), then $P \Rightarrow wp(C, Q)$ is also within FOL. In this way, Hoare logic is translated into FOL, which is a well-studied subject in mathematical logic and computer science.

As modern programming languages evolve, Hoare logic has also gone through various adaptations, with support for heap reasoning [143], concurrency reasoning [29], modal reasoning [75], and more. Despite the variations, the deductive principles of Hoare logic and weakest pre-condition calculus remain central to program verification.

Around the time Hoare logic was introduced, mechanized theorem provers also began to emerge. Early examples include LCF [135] and Nqthm [27] in the 1970s. Although these provers were not exclusively designed for program verification, they laid the groundwork for computer-aided formal reasoning. For instance, Nqthm was succeeded by ACL2 [89, 90], while LCF gave rise to HOL [65], which remains as a part of Isabelle/HOL [125]. The field evolved into interactive theorem proving (ITP) as we know it today, with other popular tools such as Coq [18], Lean [117], and PVS [130], often used for program verification.

The field of automated theorem proving (ATP) branched off from the early day provers. While ITP frameworks can handle general-purpose reasoning under user guidance, ATP tools focus more on fully automated first-order theorem proving. TPTP [153], a library of ATP problems, along with ATP tools such as Otter [109] and Vampire [165] were developed in the 1990s. ATP is not as widely used for program verification purposes, but many key ideas, such as resolution, superposition, and axiom selection remain relevant.

In parallel, boolean satisfiability (SAT) solvers have been evolving since the 1960s [45].

[1] To be more precise, the relation described by the Hoare triple is known as *partial correctness*, where termination is proven separately.

SAT formulas are in propositional logic, which, while less expressive than FOL, remains a foundational concept in computer science [87]. Around the 2000s, SAT solving saw breakthroughs in terms of efficiency and scalability, with new CDCL-based solvers such as GRASP [106], Chaff [116], and MiniSat [55].

Building on top of the success of SAT solvers, Satisfiability Modulo Theories (SMT) solvers also gained traction. SMT is a generalization of SAT, extending the formulas to FOL. While ATP also handles FOL problems, SMT emphasizes the integration of background theories such as arithmetic, arrays, and bit-vectors. Due to the high degree of automation and expressivity, SMT solvers such as Z3 [47], cvc5 [10], and Yices [54] have become popular in a wide range of applications, including model checking [41, 92, 111], symbolic execution [30, 31, 148], and program synthesis [4, 68].

SMT-based automated program verification (APV) also became practical with the increasingly more powerful solvers. There have been continuous developments in APV tools, including Boogie [11], Chalice [98], and VCC [146] in the 2000s; Dafny [97], Why3 [22], F* [154], Viper [118], JayHorn [85], Ivy [132], Vale [23], Nagini [56], and Prusti [8] in the 2010s; RustHorn [108], GoBra [169], Creusot [50], Flux [95], and Verus [94] in more recent years.

This concise recapitulation of the history of program verification is by no means exhaustive, but it helps to understand the current state of the field. Nowadays, deductive program verifiers generally fall into either ITP-based or APV-based. ITP, while not the focus of this thesis, has an earlier history, with a persistent impact in the formal methods community. At a very high level, ITP tools such as Coq or Lean enable developers to construct proofs interactively. Meanwhile, APV tools such as Dafny or Verus partially automate the process, where the developer provides some proof annotations as hints. Here we compare and contrast the two approaches in more detail.

User Interface. With ITP, a developer can examine the proof state through a live "proof assistant" window, which presents the proven statements and the current sub-goals. This is not the case with APV, where the developer basically receives an overall pass/fail result. For more detailed information, e.g., which sub-goal remains unsolved, one often needs to adjust the proof annotations and re-run verification. We discuss the iterative development process in APV with greater detail in Sec. 2.3.

Proof Automation. In ITP, the developer writes tactics to manipulate the proof state. A tactic is similar to a step in a pen-and-paper proof, e.g., applying a lemma, or simplifying an expression, but beyond that, a tactic can also programmatically orchestrate a proof, e.g., applying another tactic repeatedly until a sub-goal is solved. In a sense, tactics create a means to program the proof search. Therefore, the degree of automation depends on the tactic library available and the developer's expertise. In APV, the automation is more "uniform", where the proof obligations are encoded as SMT queries and handed off to solvers. There is no direct way to guide the proof search in APV. The developer can only do so indirectly by adjusting the proof annotations.

Language Embedding. The Hoare logic of some language L is based on the syntax

and semantics of L. In ITP, one usually defines L as a library. For example, using Coq, CompCert [99] embeds a subset of C, and RustBelt [84] embeds a subset of Rust. In APV, the language L is often embedded in the verifier itself. For example, Verus is an APV tool for Rust, where the semantics of Rust is hard-coded into the verifier. Similarly, Dafny is an APV tool for itself, where the executable subset of Dafny is similar to C#. It is feasible to embed another language in an APV language (as we do in Chapter 5), but less common.

2.1.2 Basic Concepts in Logic

A deductive system consists of the following components:

- **Syntax.** The set of well-formed formulas[2].
- **Semantics.** The definition of truth (validity), usually in terms of models.
- **Axioms.** A set of formulas assumed to be valid.
- **Proof Rules.** The rules for building new formulas (theorems) from existing ones.

The logical symbols, which include connectives (e.g., $\wedge, \vee, \neg$), quantifiers ($\forall, \exists$), constants ($\top, \bot$), and equality ($=$) usually just have the standard semantics. Function symbols, e.g., a binary operator $+$, can have interpretation-dependent semantics. Specifically, in equality logic with uninterpreted functions (EUF), non-logical symbols have no meaning, where $+$ can simply represent a deterministic binary function.

An interpretation $\mathfrak{M}$ assigns meaning to the non-logical symbols. If a formula ϕ is true under $\mathfrak{M}$, $\mathfrak{M}$ is said to be a *model* of ϕ. For example, consider a propositional formula $a \wedge b$, where a, b are uninterpreted boolean functions (i.e., unknown boolean constants). One interpretation is $\{a \mapsto \top, b \mapsto \top\}$, which is a model, but $\{a \mapsto \top, b \mapsto \bot\}$ is not.

We now restrict possible interpretations to models of the axioms. (If the axioms have no model, the logic is inconsistent.) A formula ϕ is *valid* iff ϕ is true under all interpretations. A formula ϕ is *satisfiable* if ϕ has at least one model. A formula ϕ is *unsatisfiable* if ϕ is false under all possible interpretations.

We note unsatisfiability is a dual to validity (rather than satisfiability). If we negate a valid formula, the result is unsatisfiable, and vice versa. However, if we negate an satisfiable formula, the result can be either satisfiable or unsatisfiable.

The set of *provable* formulas is the transitive closure of derivable theorems from the axioms, based on the proof rules. However, there is no guarantee that the provable formulas are valid, and vice versa. This dichotomy between derivability and validity leads to a few basic properties of a logic.

- **Soundness.** A logic is *sound* when all provable formulas are true.
- **Completeness.** A logic is *complete* when all true formulas are provable.
- **Consistency.** A logic is *consistent* when it is contradiction-free. A contradiction (i.e., inconsistency) manifests as ϕ and $\neg\phi$ being provable for some formula ϕ.

[2]For the rest of the discussion, we implicitly assume the formulas under consideration are well-formed.

Depending on the expressivity, a logic may differ in the following aspects:

- **Order.** The order of a logic depends on what it can quantify over. Propositional logic, sometimes called zeroth-order logic, is quantifier-free. First-order logic (FOL) quantifies over non-logical domains (e.g., integers, strings) only. Higher-order logics (HOL) quantify over functions or predicates.
- **Sortedness.** An unsorted logic has a single domain of discourse, so a function's signature is just the arity. A many-sorted logic makes it possible to quantify over individual domains. Function signatures are also become more expressive.
- **Decidability.** A logic is *decidable* if there is an algorithm that determines the validity of any formula.

First-Order Theories. Using a set of first-order formulas as axioms creates a first-order theory. A theory can be unsound or inconsistent, if an axiom is invalid or the axioms contradict each other. For practical (implementation) purposes, a first-order theory with infinite number of axioms is incomplete. For example, Peano Arithmetic (PA) is a first-order theory, but the induction principle requires an infinite number of axioms.

FOL (by which we mean quantified EUF here) is semi-decidable, so that there is an algorithm proving valid formulas, but no algorithm that can decide general invalid formulas exists. In practice, effectively checking valid FOL formulas is a difficult problem. Notably, quantifier instantiation (QI) often poses a challenge, i.e., the process of substituting the quantified variables with appropriate terms in order to reach a proof.

We have omitted some other properties such as compactness in FOL. Nevertheless, we have the listed the most relevant properties to this thesis, where Satisfiability Modulo Theories (SMT) plays a central role.

Satisfiability Modulo Theories. SMT is based on many-sorted FOL[3] extended with background theories. In a simplified view, an SMT query introduces a set of functions symbols, along with a set of logical constraints over them. The semantics of the query is satisfiability of the conjunction over the constraints. Excluding basic errors conditions (e.g., due to syntax), the possible outcome of an SMT query is one of the following: `sat`, `unsat`, `unknown`, and `timeout`.

SMT is many-sorted, where each term is tied to a fixed sort. The sort system enforces disjointedness over objects of different sorts, which is similar to type systems in programming languages. However, the sort system is somewhat limited in expressivity. For example, SMT-LIB offers very limited support for polymorphic sorts, while polymorphism is a common feature in programming languages.

SMT defines the signatures for a range of background theories, including arithmetic, arrays, and bit-vectors. These theories are in the "background", as in they do not have explicit, callable first-order axioms. Nevertheless, the sorts and functions symbols from

[3]As of writing, SMT-LIB 2.7 [15] has very recently introduced the standard for polymorphic sorts and HOL. However, the new standard is not yet widely adopted/implemented. Therefore, we focus the discussion on SMT-LIB 2.6 [14], which has been in place since 2017.

supported theories (e.g., `Int` and + for integer arithmetic) have predefined semantics. In contrast, query-declared functions are uninterpreted by default. For example, the command `(declare-fun foo (Int) Int)` declares an uninterpreted function symbol `foo`. The solver knows nothing about `foo` except its signature, unless given further constraints over it.

SMT solving is generally undecidable and incomplete. While FOL is semi-decidable, many background theories in SMT are undecidable, including the theory of nonlinear integer arithmetic [107], which is fairly common in program verification. Moreover, incompleteness may also be due to certain heuristics. Notably, APV languages including Dafny, F*, and Verus rely heavily on pattern-based quantifier instantiation (QI) [46, 115], which is incomplete by design. We discuss patterns more formally in Sec. 2.4.

2.2 System Software Overview

In this thesis, we focus on the verification of system software. Generally, system software provides infrastructure to other software, as opposed to application software that serves end-users directly. Examples of system software include operating systems, file systems, network protocols, and hypervisors. It is important to note that machine learning systems and hybrid systems (e.g., cyber-physical systems) are outside the scope of this thesis, as they emphasize different program properties and thus require distinct theoretical foundations for verification.

2.2.1 Requirements for System Software

We discuss a few broad classes of requirements for system software. While these requirements are often the high-level properties that program verification aims to establish, we start from the perspective of a system software developer/designer. This is to provide a more practical view of verification, which is the focus of this thesis.

- **Functionality.** Advanced properties often become irrelevant without basic functionality. For instance, a protocol implementation that drops all messages also leaks no information about the messages. However, message secrecy here is pointless, as the system is not functioning at all.
- **Robustness.** System software is often mission-critical, so it should gracefully manage unexpected situations. Robustness might have more system-specific implications, but it generally includes handling unexpected inputs, recovering from failures, and maintaining availability. For example, a storage system should recover to a consistent state after power failure, and it should maintain data integrity in the presence of concurrent accesses.
- **Security.** System software is often security-critical. Security requirements tend to overlap with robustness requirements, but focus more on malicious threat models. For example, a storage system should prevent unauthorized data access, including

deliberate attempts to bypass the access control mechanisms, rather than accidental corruption of data due to power failure in our previous example.

- **Efficiency.** System software is often performance-critical. Performance typically refers to time-related metrics such as latency and throughput. In some cases, memory usage, power consumption, and other resources might also be of concern. For example, a cryptographic library on an embedded device might be subjected to very limited memory, As a result, minimizing the memory footprint, including the compiled binary size itself, becomes a part of the efficiency requirement.

While these requirements all contribute to the overall quality of a system, robustness or security often conflicts with efficiency. For example, data redundancy may improve the robustness of a storage system, but at the cost of additional disk space; similarly, packet encryption may improve the security of a network protocol, but at the cost of increased latency. Generally, robustness and security come with a performance cost, so system software often needs to minimize the overhead.

As a result, system software often ends up with complex low-level implementations, squeezing out the last bits of performance. Specifically, languages such as C/C++ or assembly are common in the implementations, which provides the fine-grained resource control that is essential to performance. For example, a C program can can apply struct-padding for better page alignment or reorder allocations for fewer cache misses; either would be fairly difficult to achieve in Java or Python programs. Similarly, using assembly languages allows architecture-specific SIMD or bit-manipulation tricks, which a high-level language compiler might miss.

2.2.2 Implications for Program Verification

With the general requirements for system software in place, we discuss how program verification fit into the picture. As a starting point, we adopt the perspective that verification is a means to an end, rather than an end in itself. While we do sometimes make compromises to facilitate verification, we acknowledge them as limitations of APV, which we aim to address in this thesis.

Purposes of Verification. System software is vulnerable to low-level bugs, which verification can help prevent. Specifically, *memory safety* has been a long-standing issue. Due to the low-level nature of programming languages like C/C++, programmers have to perform manual memory management and pointer arithmetic, which may introduce subtle bugs such as buffer overflows and use-after-free. These bugs often have security implications, including remote code execution and privilege escalation. Worse yet, the bugs may occur in silence[4], making them hard to detect with traditional testing methods.

In Sec. 3.1.2, we discuss memory reasoning as the first concrete challenge. We place

[4]In Java, an out-of-bounds array access would lead to an exception, but in C, it might create a segfault, or it could just corrupt memory without any error messages.

such emphasis on memory safety, not only because it is a common issue in system software, but also because it is a prerequisite for proving many other properties. Specifically, without memory safety, the program's behavior is undefined [77], making it impossible to reason about functionality, robustness, or security.

In addition to preventing low-level issues, verification can also help establish high-level properties. Specifically, it is often advisable to validate a design before investing heavily in its implementation. With programs verification, we can first prove a design model possesses the desired properties, and then prove that the implementation *refines* the model. However, unlike memory safety, high-level properties are often system-specific. For example, the security of a cryptographic protocol has a very different definition from the security of a virtual machine monitor. Thus, it often requires domain-specific knowledge to verify these advanced properties.

Expressivity of SMT. Despite the complexity of system software, first-order reasoning is usually sufficient to capture and prove the properties of interest. Intuitively, there is rarely a need to quantify over functions or predicates. For a system developer, they are more likely to start a statement with "For all nodes in this list...." than with "For all the first-order predicates over the structure....", which might be more typical for a logician. Even in the rare case where a higher-order statement seems necessary, we can often reduce it to first-order. Suppose that we would like to claim "For all the procedures in this codebase, it is memory safe". This is technically a second-order statement, since it quantifies over procedures. However, a codebase has finitely many procedures, which we can simply enumerate and write a first-order statement instead.

Meanwhile, there is often the need to augment pure FOL with background theories, which is a key feature of SMT. For instance, when reasoning about cryptographic primitives, we need to connect the bit-level manipulations to high-level algebraic properties, which requires the theory of bit-vectors and finite fields.

However, with expressivity comes the cost of decidability. Even if the code satisfies the specifications, it is perfectly valid for an SMT solver to output `unknown`/`timeout`. Intuitively, we would not expect an SMT solver to prove Fermat's Last theorem without any hints, so similar expectation applies to complex program verification tasks.

Meanwhile, the *push-button verification* style in APV offers a trade-off, with a focus on decidable logics. This approach has seen some success in system software [122, 123, 124, 131]. The obvious advantage is that full automation becomes possible, but the expressivity is limited. For example, recursive functions or loops might need to statically bounded, integers might need to have finite ranges, quantification might be subject to *Effectively Propositional Reasoning* [134], formulas might be restricted to *Constrained Horn Clauses* [69], and so on. For our discussion in this thesis, we do not assume such restrictions.

2.3 Automated Program Verification

Automated Program Verification (APV) has been a popular approach for system program verification. In this section, we offer an overview of the APV workflow and its key characteristics. Our discussion here applies to languages such as Dafny, F*, Verus, and many more, which we mentioned in Sec. 2.1.1.

As we discussed in Sec. 2.1, the verifier takes as input the executable code, specifications, and proofs. To avoid confusion, we use *source program* to refer to the entire input, and *code* to refer to the part that is meant to be compiled/executed.

2.3.1 Overall Pipeline

The APV pipeline has three main stages: **(1)** writing the source program; **(2)** generating the SMT queries, and **(3)** checking the queries with an SMT solver. The SMT query is essentially a different representation of the source program, in a form that is amenable to automated reasoning by the SMT solver. Since we already have introduced the preliminaries of SMT in Sec. 2.1.2, we first discuss the source program here, and then how an APV language bridges between the source and the SMT.

Source Procedure. Procedural abstraction is central to the modularity of Hoare-style verification, where we can verify each procedure independently, while assuming the correctness of the others as axioms. We generically refer to a function-like construct with pre/post-conditions as a procedure. It can be a function, method, lemma, etc. In certain contexts, it is worth making a distinction between *executable* and *ghost* procedures, where the latter is for specification and proof purposes only.

Proof Annotations. Proof annotation is a key language feature in APV. Programming languages often have `assert` statements that dynamically enforce user-defined properties. APV languages also have `assert` statements, which statically state the properties instead. That is, the `assert` statements in APV would fail if the property cannot be statically verified. The terminology unfortunately overlaps with the `assert` command in SMT-LIB. To make a distinction, we refer the source-level statement as *proof annotations*, or simply annotations; and the SMT-level commands as *assertions*. We also make a distinction between annotations versus pre/post-conditions, or invariants, which are all *specifications*.

An annotation can be interpreted in two ways: **(1)** as an obligation to be proved and **(2)** as an intermediary step towards other proof obligations. Consider the Dafny lemma in Lst. 2.1 that proves two polynomials equal. The annotation on line 3 is an obligation to be proved, but it can also bridge the gap towards the post-condition on line 1.

The proof annotations do not have to be "complete": they can skip some reasoning steps. The example lemma is missing the steps on line 4 and 5 but can still be verified. This is due to the automation provided by SMT solvers, which can often power through the proof despite not having all of the details.

```
lemma foo(a:int, b:int, c:int)
  ensures (a+b)*c == c*a + c*b
{
  assert (a+b)*c == a*c + b*c;
  // assert a*c + b*c == a*c + c*b;
  // assert a*c + c*b == c*a + c*b;
}
```

Listing 2.1: **Example Dafny Program**

Ghost Variables. Similar to how executable code contains compiled variables, proofs and specifications may also contain auxiliary ghost variables. In fact, our example lemma in Lst. 2.1 contains ghost variables only, since a lemmas is a ghost procedure with no runtime semantics.

Ghost variables, when present in a executable procedure, might require additional type declarations, e.g., with the `ghost` keyword as a type modifier in Dafny. Ghost variables may also augment compilable data structures as members, which can be quite useful for specifying data structure invariants. At compile time, all ghost variables or procedures are erased from the source program.

Verification Conditions. The main responsibility of an APV language is to translate source programs into SMT queries. The process is often called *verification condition generation* (VCG), which implements the weakest precondition calculus we mentioned in Sec. 2.1.1. The VCG typically works on a per-procedure basis, translating a given procedure's code, specification, and proof (if any) into a verification condition [51]. Informally, the verification condition, which we call the *goal*, is a logical formula stating that the code satisfies the specification for the given procedure. Overall, verification of a project is equivalent to proving the verification goals from all its member procedures.

More specifically, the VCG emits the *negated* version of the goal into the SMT query. The intuition is that if this query is unsatisfiable, then the specification is never violated, and thus the procedure verifies. Otherwise, the verifier (e.g., Dafny) reports the verification failure to the programmer, similar to how a compiler reports errors and warnings.

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(assert
  ; negates the verification goal
  (not
    (let (
      ; corresponds to the post-condition line 1 Lst.1.1
      (Q (= (* (+ a b) c) (+ (* a c) (* b c))))
      ; corresponds to the proof annotation line 3 Lst.1.1
      (A (= (* (+ a b) c) (+ (* c a) (* c b)))))
      (and
        ; A should hold
        A
        ; given A, Q should hold
        (=> A Q)))))
(check-sat)
```

Listing 2.2: **Example SMT Query**

In Lst. 2.2, we show one possible SMT encoding of the lemma. We use an `assert` command to introduce a constraint, which negates the verification goal. In the goal, we use let bindings to define the proof annotation A along with the post-condition Q, and then state $A \wedge (A \Rightarrow Q)$. Intuitively, Q could be easier to establish if we first prove A.

Context Assertions. For simplicity, we used an example with a single assertion in the SMT query. More generally, the VCG may also insert additional assertions into the query context. Specifically, for a source procedure under verification, the VCG may need to axiomatize the semantics of: **(1)** the language constructs, **(2)** the standard library functions, and **(3)** the procedures that the current procedure calls. In practice, these may translate into thousands of additional assertions.

Determining the callee set in **(3)** can be challenging. For instance, when higher-order functions or dynamic dispatch are involved, it may be impossible to statically determine the precise set of callees. To address this, the VCG usually over-approximates the call graph. This approach ensures completeness, as under-approximation might preclude the verification of correct programs. However, over-approximation may introduce irrelevant assertions, potentially leading to solver performance issues, which we explore further in Sec. 6.2.

The callee set also brings up the notion of *visibility*. In particular, for a procedure A under verification and another procedure B in the same project, there are three common visibility levels:

- **Signature.** A can see B's signature, but not its implementation (body).
- **Full.** A can see B's signature and its implementation.
- **None.** A cannot see B at all.

In the special case when B is recursive, there is also the notion of *fuel*, which is the maximum number of B can be expanded in A.

APV languages usually offer source-level language features to override the default visibility level defined by the VCG. For example, a Dafny `function` has full visibility by default (within the same module). Meanwhile, Dafny also has the `opaque` and `reveal` keywords, which hides the function body, and only reveals it when explicitly requested at a call site. Similar mechanisms exist in F* and Verus.

2.3.2 Development Workflow

In this section, we discuss the software and proof engineering aspects of APV. This will hopefully provide some perspective on the scalability challenges we cover in this thesis.

The lifecycle of an APV project is similar to that of traditional software, starting from conception, to development, to maintenance, and finally to retirement. However, since a lot of the APV projects are academic research projects, the maintenance phase is often short-lived. As a result, the development phase has been the main focus of research.

2.3.2.1 Development Concerns

During active development, the developer typically focuses on a small portion of the source program (often scoped to a single procedure). The process usually follows an iterative workflow:

(1) The developer runs the verifier on the current procedure to check if it passes.
(2) If not, they add some debugging annotations.
(3) If so, they make more concrete changes, e.g., adding executable code.
(4) Either way, they go back to step (1), re-running the verifier.

The procedure-oriented, iterative process has a few implications in practice:

Time. The developer is blocked while the solver is running, so the procedure-level verification time should be in the responsive range of human interaction. In particular, Verus has a 10-second limit[5] by default. In older (now legacy) projects written in Dafny and F*, a 60-second limit is common.

Feedback. Feedback from the verifier (i.e., the SMT solver) is of particular importance, especially when the verification fails and the developer needs to understand the cause. Unfortunately, the SMT solver is mostly a black box, and obtaining information from it is a somewhat arcane art. We thus dedicate Chapter 4 to annotation-based proof debugging, where we discuss the common debugging techniques in step (2). We also explore the possibility of making the process more transparent in Chapter 6, leveraging various log files provided by the SMT solver.

Locality. Despite the procedural abstraction, a local change can have a wider impact on the verification results. In particular, changing a procedure may require re-verifiying all its callers, if not the transitive callers. Therefore, a change to a procedure impacts not only the SMT queries for the procedure itself, but also those for all its callers, potentially propagating throughout the call graph.

Stability. As we introduced in Sec. 1.2.4, instability is the phenomenon where minor, non-semantic updates cause unexpected verification failures (which sometimes are not even local to the change). For example, renaming a source-level variable might lead to a previously verified procedure taking significantly longer to verify or even failing entirely. In such scenarios, developers often need to provide additional proof hints to help the SMT solver efficiently and successfully complete the verification process.

Instability thus poses a significant challenge for large-scale, industrial-level APV projects. For a developer, instability disrupts the normal workflow and substantially lengthens their iterative development cycles. Moreover, spurious failures may require developers to fix proofs/code 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

[5]Verus actually bounds the solver execution using resource count (i.e., `rlimit` in Z3), where the default resource limit corresponds to ~10 seconds of CPU time.

monotonic progress in developing a verified codebase.

Worse yet, instability is deeply rooted in the nature of APV. As we discussed in Sec. 2.2.2, system software verification often involves a combination of complex low-level implementations and undecidable high-level properties. Therefore, the solver inevitably resorts to incomplete heuristics, causing it to give up on certain queries. Hence, the developer is left with spurious verification failures that seem to be unrelated to the changes they made. Nevertheless, the problem is not as hopeless as it seems. We devote Chapter 6 to the study of instability, where we explore the causes and potential solutions.

2.3.2.2 Maintenance Concerns

While the active development phase is often the focus of research, as APV moves towards real-world applications, the maintenance phase is also receiving more attention. There are roughly three types of maintenance updates due to: **(1)** changing requirements or design decisions; **(2)** refactoring for better organization and readability; and **(3)** verifier upgrades or changes in the underlying SMT solver.

We note that **(1)** and **(2)** also apply to the active development phase, but maybe the scope of the change tends to be larger in the maintenance phase. For example, renaming a file from `Foo.dfy` to `Bar.dfy` might have a larger impact than renaming a local variable from `x` to `y`, where the latter is more common in active development. Nevertheless, our discussion on instability remains highly relevant, if not more so, in the maintenance phase.

It is worth pointing out that **(3)** tends to be rather rare in practice, but points to an orthogonal issue from instability. For example, HACL* [137], based on F* (or rather, Low* [141]), has been a long-lived project since 2017 [177], and it is still under active maintenance as of 2025. While F* has evolved over time, the APV language "pinned" itself to Z3 version 4.8.5 (also released in 2017) for a long time. Anecdotally, when F* finally upgraded to Z3 version 4.12.1 in 2023, there were a significant number of regressions, which we similarly observed on other verification projects (Sec. 6.1.4.4).

2.3.3 Common Techniques

In this section, we discuss a few common techniques in program verification. Due to how general they are, they also come up a few times in this thesis, so it might be helpful to introduce them here.

Data Structure Invariants. We usually do not burn down a bridge after crossing it — if we ever want to go over it again. The same idea applies to data structures, which we often do not immediately abandon after updates. To show our appreciation for a data structure, we can define its specifications in the form of an *invariant*, which we (ghostly) maintain as we make updates to the structure. Let us demonstrate the paradigm with a bridge:

(1) We specify the invariants, e.g., "the bridge is not on fire, and it is safe to cross if it is not on fire."
(2) We show that the operation is sane under the invariants, e.g. "given the bridge invariant, we can cross safely."
(3) We show preservation after the update, e.g. "we crossed the bridge, resisted our arsonistic urges, so the bridge invariant still holds."

Meanwhile, it might be entertaining to come up with cases where destroying the invariant is the better thing to do. For example, when we are about to exit a program, skipping heap deallocation violates memory safety, but would be practically faster.

Refinement Relation. A refinement relation connects a abstract model to its more concrete counterpart, typically through an interpretation function $\mathcal{F}$. In fact, many data structure invariants are refinement relations, where an operation on the abstract model is realized by an operation on the implementation.

For example, consider the `BigInteger` class (e.g. in Java or C#), which often shows up in cryptographic libraries. `BigInteger` typically contains some array of machine words as its private class member. The interpretation function $\mathcal{F}$ maps this array to a mathematical integer, answering design questions such as:

- Where is the sign bit?
- What is the weight of each word?
- Is it big or little-endian?

With a reasonable design, we should be able prove that the refinement relation remains invariant. For example, our array-based addition $\oplus$, multiplication $\otimes$, and subtraction $\ominus$ should satisfy the following properties:

$$\begin{aligned}\mathcal{F}(x \oplus y) &= \mathcal{F}(x) + \mathcal{F}(y)\\ \mathcal{F}(x \otimes y) &= \mathcal{F}(x) \times \mathcal{F}(y)\\ \mathcal{F}(x \ominus y) &= \mathcal{F}(x) - \mathcal{F}(y)\end{aligned}$$

In a sense, refinement in verification is akin to homomorphism in algebra. Similar to homomorphisms, we can also compose refinements, which is useful for the separation of concerns in a complex system. For example, we might decompose a distributed key-value store into a pure functional layer specifying the overall behavior, a protocol layer describing the messages between nodes, and an implementation layer sending packets over the network. This pattern with multiple layers of refinement is sometimes called the *refinement stack*, which we use in Chapter 5, in the context of verified assembly code for cryptographic primitives.

2.4 Notations and Terminologies

In this section, we introduce the notations for FOL and SMT we use in the thesis. This part is of particular importance to Sec. 6.2 and Sec. 6.3, where we discuss proof instability.

Conventions for Common Symbols.

First, we cover some conventions we assume, which are relatively standard in the literature. Unless otherwise specified, we consistently use the following:

- $\bot, \top$ for logical false and true
- x, y, z for local bound (or free) variables
- t, u, v for ground terms (i.e., terms without free variables)
- f, g, h for global uninterpreted functions
- a, b, c for global boolean constants
- n, m for natural number constants (e.g. number of assertions)
- i, j, k for non-negative integer indices
- ϕ, φ, ψ for logical formulas
- Γ, Λ for sets of logical formulas

We use the word *constant* for fixed but arbitrary values, i.e., 0-arity uninterpreted functions. We use the notation $\{\cdot\}$ for a set of elements. We use $\|\cdot\|$ to denote the cardinality of a set, along with the union and intersection operators $\cup$ and $\cap$. We use $\setminus$ to denote the set difference operator. For example, $\Gamma \setminus \{\psi\}$ is the set of elements in Γ except for ψ. We use $\langle\cdot\rangle$ to denote an ordered sequence of elements, and $\|\cdot\|$ for the length of the sequence as well. For example, $\|\langle\psi_0, \psi_1, \psi_2\rangle\| = 3$.

Notations for Logical Formulas.

We reserve $\cong$ for the equality operator within logical formulas, differentiating it from the meta-level equality ($=$) in our English description. For example:

- $\varphi = (\forall x. f(x) \cong g(x))$ is a valid notation.
- $\varphi \cong (\forall x. f(x) \cong g(x))$ is also a valid notation.
- $\varphi = (\forall x. f(x) = g(x))$ is not well-formed.

In the first case, we associate φ with $\forall x. f(x) \cong g(x)$ so that φ is a meta variable referring to the quantified formula. In the second case, φ is a part of the formula itself, rather than a meta variable.

We use $\varphi[x \mapsto t]$ to denote the result of capture-free substitution of the ground term t for free variable x in the formula φ. We use $\sqsubseteq$ for the sub-formula relation. For example, let $\varphi = x \wedge b$, where x is free, then $\varphi[x \mapsto t] = t \wedge b$, and $t \sqsubseteq \varphi[x \mapsto t]$.

We abstract an SMT query as the conjunction of its assertions, using the upper case Φ to denote a query:

$$\Phi = \bigwedge_{i=0}^{n} \psi_i$$

where ψ_i is the i-th assertion. Since the order in which we introduce assertions has no impact on the satisfiability of Φ, without loss of generality, we *assume* that:

- Φ is already in conjunctive normal form (CNF).
- Φ is non-empty, and ψ_0 encodes the verification goal.
- The assertions are de-duplicated.

We use Γ_Φ to denote the set of assertions comprising the query *context* in Φ:

$$\Gamma_\Phi = \{\psi_0, ..., \psi_n\}$$

Since we assume that ψ_0 encodes the goal, we refer to the rest of the context as the *axioms*:

$$\Lambda_\Phi = \{\psi_1, ..., \psi_n\}$$

We may use the term assertion or axiom interchangeably when we refer to some $\psi_i \in \Gamma_\Phi$. However, when we use the phrase *axiom* ψ_i, we implicitly mean $\psi_i \in \Lambda_\Phi$. We note that while $\Gamma_\Phi \vdash \bot$ is expected, Λ_Φ is supposed to be contradiction-free.

We make a note of the context subset relation $\Gamma_{\Phi_0} \subseteq \Gamma_{\Phi_1}$ between two queries Φ_0 and Φ_1. In particular, an *unsatisfiable core* of Φ, which we denote by Φ_C, is formed by a subset of Γ_Φ sufficient to derive a contradiction. More formally, Φ_C has the following properties: $\Gamma_{\Phi_C} \subseteq \Gamma_\Phi$, $\Lambda_{\Phi_C} \subseteq \Lambda_\Phi$, and $\Gamma_{\Phi_C} \vdash \bot$.

The use of quantification is common in APV queries. For the ease of exposition, we use *single-variable* quantified formulas as examples as long as it is clear how the method under discussion generalizes to quantification over multiple variables. We use the lower case ϕ to denote a quantified formula, and φ to denote the *quantified body*, e.g., $\phi = \forall x.\varphi$.

In our terminology, a quantified formula ϕ must directly start with a quantifier. For clarification, we introduce the following helper functions:

- IsForall(ϕ) when $\phi = \forall x.\varphi$ for some φ.
- IsExists(ϕ) when $\phi = \exists x.\varphi$ for some φ.
- IsQuant(ϕ) when either IsForall(ϕ) or IsExists(ϕ) is true.
- HasQant(ψ) when there exists $\phi \sqsubseteq \psi$ such that IsQuant(ϕ).
- QantFree(ϕ) when HasQant(ϕ) is false.

We note the difference between IsQuant and HasQant. For example, let $\psi = a \vee (\forall x.\varphi)$. In this case HasQant($\psi$) is true, but IsQuant($\psi$) is false.

We define the set of quantified formulas in Φ as:

$$\Omega_\Phi = \{\phi \mid \phi \sqsubseteq \Phi, \text{IsQuant}(\phi)\}$$

we note that Ω_Φ further includes all *nested* quantified formulas as its elements, which are not directly present in Γ_Φ or Λ_Φ. As a summary, we have the following terminologies for a formula ϕ when IsQuant(ϕ):

Given $\phi = \forall x.\varphi$ and a ground term t, we refer to $I = \varphi[x \mapsto t]$ as an *instantiation* of ϕ, and t as the *instantiating term*. For our purposes, we only have to consider instantiating

Term	Scope
quantified axiom	$\phi \in \Lambda_\Phi$
quantified assertion	$\phi \in \Gamma_\Phi$
quantified formula	$\phi \in \Omega_\Phi$

Table 2.1: **Terminologies for Quantified Formulas**

terms that are ground. Given an existentially quantified formula $\phi = \exists x.\varphi$, we use f_{ϕ_x} to denote the Skolem constant (function) for the bound variable x in ϕ, and the result after Skolemization as $\varphi[x \mapsto f_{\phi_x}]$.

The APV queries we study often rely on pattern-based [115, 121] quantifier instantiation (QI). For a universally quantified formula, the verification language or the developer may attach one or more syntactic patterns. Consider single-variable quantified formula $\phi = \forall x.\varphi$ with some pattern π. The pattern π would be an ground term otherwise, except that x (bounded by the quantifier) is free in π. The quantified body φ remains hidden until a ground term $u = \pi[x \mapsto t]$ is discovered, i.e., the pattern π matches against u with the substitution $\{x \mapsto t\}$, at which point the instantiation $\varphi[x \mapsto t]$ is created.

```
0  (declare-fun foo (Int) Int)
1  (declare-fun bar (Int) Int)
2  (declare-fun qux (Int) Int)
3  (assert (forall ((x Int))
4                  (! (< (foo x) (bar (qux x)))
5                     :pattern ((foo x))
6                     :pattern ((bar x)))))
7  (assert (= (bar (qux 2)) 3))
8  ; triggers (bar x), effectively introducing:
9  ; (assert (< (foo (qux 2)) (bar (qux 2))))
10 (assert (= (qux 2) 4))  ; will not trigger
```

Listing 2.3: **Example SMT Quantified Axiom with Pattern(s)**

In Lst. 2.3, we have a quantified axiom with two patterns `(foo x)` and `(bar x)`. The assertion on line 7 contains the term `bar (qux 2)`, which triggers the pattern `(bar x)` with the substitution `x` mapped to `(qux 2)`. This would effectively introduce the assertion on line 9 (commented out). Pattern-based QI is incomplete (by design). For example, the assertion on line 10 does not trigger either pattern, and thus does not introduce new instantiations. Pattern-based QI generalizes to multiple variables, where a pattern needs to contain all the variables bounded by the quantifier.

Notations for Runtime Behavior.

We have been defining notations for the more theoretical concepts so far. We now introduce notations that capture the execution of a solver s on a query Φ. That being said, we omit s from the notation as long as there is no ambiguity. For instance, in Sec. 6.2, we are interested in *some* unsatisfiable core Φ_C produced by some solver s, but which specific s to use is irrelevant. Meanwhile, it is important to note that the concepts under discussion are tied to concrete execution. For example, in theory, a query Φ can have multiple unsatisfiable cores. In practice, since solver execution is deterministic, once we have fixed a solver s and

its configurations, there is a unique solver-produced Φ_C with a given Φ.

Other than the core, we leverage two other log files from the solver, namely the trace log $\mathfrak{t}$ and the proof log $\mathfrak{p}$. While there is no standard format for either type of log, we are particularly interested in a solver's quantifier instantiation (QI) reasoning, which these logs record. Informally, a trace log contains all the quantifier instantiations that the solver has discovered when it attempts to solve Φ, while a proof log contains the instantiations that the solver has used to construct a proof tree (of unsatisfiability).

Before we proceed to the formal definitions, we need to clarify the notion of *determinism* a bit. More precisely, SMT solvers are deterministic given the same input and configuration. For example, SMT solvers support seed-based randomization, but the execution is deterministic given the same seed. However, changing the configuration (e.g., the seed) may lead to vary different results, which is not considered as a non-determinism bug. That also means enabling any type of logging changes the execution.

We use the calligraphic $\mathcal{I}^\phi$ to denote a set of instantiation for a (universally) quantified formula $\phi = \forall x.\varphi$. For example, $\mathcal{I}^\phi = \{I_1, ..., I_m\}$ has m elements, where $I_i = \varphi[x \mapsto t_i]$ for a set of instantiating terms $\{t_1, ..., t_m\}$. Intuitively, we define the *instantiation count* of a quantified formula ϕ to be the cardinality $\|\mathcal{I}^\phi\|$. For simplicity, we define $\mathcal{I}^\phi = \emptyset$ whenever IsExists(ϕ), so that $\mathcal{I}^\phi$ is well-defined as long as $\phi \in \Omega_\Phi$.

We denote the the trace/proof instantiation set of ϕ in the trace/proof log as $\mathcal{I}^\phi_\mathfrak{t}$ and $\mathcal{I}^\phi_\mathfrak{p}$ respectively. For our analysis on solver execution, these sets are always finite. In theory, it should be the case that $\mathcal{I}^\phi_\mathfrak{p} \subseteq \mathcal{I}^\phi_\mathfrak{t}$ for all the universally quantified formula ϕ in Φ. In practice, since the solver execution changes due to configuration changes, the instantiation sets may not be comparable. Nevertheless, it is generally true that $\|\mathcal{I}^\phi_\mathfrak{p}\| < \|\mathcal{I}^\phi_\mathfrak{t}\|$, and often the difference is several orders of magnitude.

We often refer a trace or a proof as an *instantiation profile*, which is essentially a map from quantified formulas to their instantiation sets.

$$\{\phi \mapsto \mathcal{I}^\phi \mid \phi \in \Omega_\Phi\}$$

Intuitively, we define the instantiation count of a profile to be the summation of the individual instantiation counts.

Chapter 3

Developing Proofs

In this chapter, we discuss the challenges when we create (new) proofs in APV projects. We mainly focus on memory-safety and theory-specific proofs. In layman's terms, this chapter is actually more about how to *avoid* these proofs, rather than how to develop them. The rationale is hopefully clear — if we can let the SMT solver do the dirty work, we should really take advantage of it.

Meanwhile, this chapter exists because the SMT solver often fails to automate the reasoning steps. As discussed in Sec. 2.2.2, memory safety is a fundamental requirement of low-level systems. However, encoding proof obligations of memory safety as SMT queries might mean poor solver performance (Sec. 3.1.1). We then expand on the concrete challenges in our QUIC$_D$, an implementation of the QUIC network protocol using Dafny (Sec. 3.1.2).

With the potential of arbitrary aliasing, memory reasoning is indeed difficult in theory. However, we observe that the points-to relations are often much simpler in practice, even in systems like QUIC$_D$. We discuss how to leverage the observation in Linear Dafny (Sec. 3.1.3), where the type checker enforces the common case linear ownership, so that the SMT solver can handle the nonlinear cases when necessary. We then evaluate how linearity improves our verification experience in VeriBetrKV$_L$, a large-scale verified key-value store (Sec. 3.1.4).

In the second part of the chapter (Sec. 3.2), we discuss theory-specific reasoning. As we discussed in Sec. 2.2.2, while the core quantified EUF is the basis of program verification, system software often involves theory-specific operations. In particular, we explore challenges and solutions for bit-vector reasoning in Sec. 3.2.1 and nonlinear arithmetic reasoning in Sec. 3.2.2.

3.1 Proving Memory Safety

Low-level memory access is often necessary to high-performance system programming. However, low-level heap management is error-prone, creating problems such as dangling pointers and buffer overflows. The good news is that program verification can eliminate these problems. We even can follow the invariant paradigm in Sec. 2.3.3, which gives us a to-do list:

(t1) Specify the expected memory layout as data structure invariants.
(t2) Prove that memory operations are safe under the invariants.
(t3) Prove that memory operations preserve the invariants.

It turns out that aliasing relations can be complex, making these tasks nontrivial. If we assume the general case, where arbitrary aliasing can occur, we essentially have to specify the points-to relations between all pairs of objects. Consider the following example in Lst. 3.1. We have two arrays, which are heap objects in Dafny. As we update one of the arrays, we cannot conclude (either way) whether the other is also affected. If we would like to show that the updates are independent, we need to explicitly specify that the arrays are distinct; if we intend the two to alias, we also need to specify that they are referencing the same array.

```
0 method mut_arrays(a1:array<int>, a2:array<int>)
1   requires a1.Length >= 10 && a2.Length >= 10
2   // requires a1 != a2
3   modifies a1, a2
4 {
5   a1[5] := 100;
6   a2[5] := 200;
7   assert a1[5] == 100; // FAIL: needs a1 != a2
8   assert a1[5] == 200; // FAIL: needs a1 == a2 (if that is what we intend)
9 }
```

Listing 3.1: **Array Update Example**

To be more precise, `a1` and `a2` are references to the arrays, which are heap objects. Meanwhile, there is no way to access the arrays other than via the references, so we simply say that `a1` and `a2` are arrays. We note that Dafny is type safe, so adding an offset to an array object reference does not type check, and thus `a1` and `a2` either refer to the same array object or are completely distinct arrays without any partial overlap.

The exact specification/proof is dependent on the formalism. A common choice is separation logic [143], which unfortunately, SMT solvers do not provide good support for. (At least there is no concrete evidence that solvers with built-in separation logic theory scale to practical systems verification.) Instead, verifiers like Dafny resort to dynamic frames [88]. At a high level, dynamic frames reduces memory safety reasoning into set reasoning, where it is fairly straightforward to axiomatize set operations into SMT.

3.1.1 Framing a Heap of Trouble

The theory of dynamic frames models the heap as a global map of symbolic memory addresses to objects (or their fields). Each procedure needs to declare its *frame*, i.e., the set of addresses that it is allowed to read/write. Each data structure, in addition to the compilable class definition, also maintains a ghost variable footprint (often called *repr*), which is a set of locations the structure has access to.

Under the formalism, memory safety properties are just predicates over sets of addresses. Let us illustrate with the invariant paradigm. For task (t1), we can now specify the points-to relation between objects through the intersecting/disjointedness of footprint sets. For task (t2), we can prove safety by showing the sufficiency of a procedure's framing condition, e.g., for every object o that the procedure accesses, the procedure's frame is a superset of o's footprint. For task (t3), depending on the memory operation, we might need to update the footprint sets, making sure that each still faithfully represents the set of locations its object can access.

In this fashion, memory reasoning is now reduced to set reasoning. Moreover, since sets of locations are first-order concepts, it is fairly straightforward to axiomatize their operations and relations into SMT. Therefore, the theory of dynamic frames has a rather simple translation from memory safety obligations into SMT queries.

However, the ease of encoding is not without a cost. First, dynamic frames essentially require the developers to describe the aliasing relation over all pairs of mutable objects, which quickly becomes complicated with deeper data structures. Furthermore, SMT solvers are not particularly good with automating the safety proofs. There is no magic in the encoding — frame reasoning is set reasoning, which translates to quantifier reasoning in SMT. Overloading the solver with a large number of quantified formulas leads to poor performance and confusing error messages when a proof fails.

3.1.2 Verifying it Real QUIC

To better illustrate the challenges of frame-based memory reasoning, we discuss our experience in implementing the QUIC protocol logic with Dafny. To clarify, QUIC is a fairly complex protocol, and our implementation QUIC_D is verified for memory and type safety (and very basic functional correctness). Therefore, the difficulties we discuss here are almost entirely related to memory reasoning, rather than the protocol-level properties.

QUIC is a network protocol that combines features from TCP (fragmentation, retransmission, reliable delivery, flow control) and TLS (key exchange, encryption and authentication) into a more integrated protocol. These roughly correspond to two components in a concrete implementation: the TCP-like protocol layer, and the TLS-like record (cryptographic) layer. The record layer has its own challenges in terms of verification, but its memory usage is very limited.

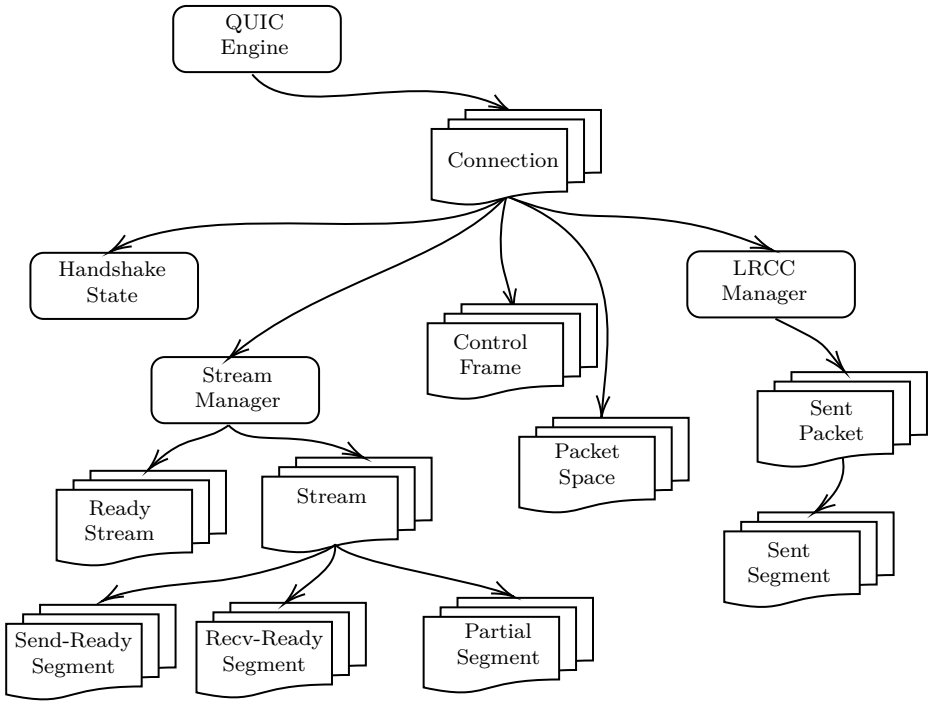


Figure 3.1: **Main Data Structure in QUIC$_D$**

Our implementation conforms to the IETF QUIC standard [93] (draft version 30 at the time of the work). More specifically, QUIC$_D$ provides the following functionalities to applications:

- Open a connection as a client.
- Listen for connections as a server.
- Control and configure various resources (e.g., number of permitted streams).
- Open/close streams in a connection.
- Write to/read from a stream in a connection.

We now briefly describe the data structures implementing the functionalities in QUIC$_D$. In Fig. 3.1, we present a simplified version of the class hierarchy, where use the stacked-documents shape to represent a list of objects of the same type.

As the interface to applications, our top-level object is `Engine`, which is either a client or a server. In server mode, we maintain a list of `Connection` objects in `Engine`, interacting with multiple clients. To ensure reliable delivery, we store the additional connection states for loss recovery and congestion control in `LRCC`. To support stream multiplexing, we maintain multiple `Stream` objects per connection. Under each stream, we keep several lists of `Segment` objects, where a segment is essentially a chunk of packet data.

The actual implementation is more involved, which means the frame specifications can get quite complex, let alone the proof obligations. In Lst. 3.2, we show the `reads` frame of a procedure in the `Connection` class, with 16 lines dedicated to specifying the addresses

we can read from. (We haven't even included the `modifies` frame!) We note how each member of a class has a referenceable location, e.g., `lrcc_manager` and `stream_manager`. This level of fine-granularity in the frame specification is necessary, since the members can be independently accessed.

```
0  reads this;
1  reads short_header_packets, // UNIQUE: Vector<packet_holder_fixed>
2      short_header_packets.buffer; // UNIQUE: array<packet_holder_fixed>
3  reads fixedframes, //  UNIQUE: Vector<fixed_frame_fixed>
4      fixedframes.buffer; // UNIQUE: array<fixed_frame_fixed>
5  reads lrcc_manager, // UNIQUE: LRCCManager
6      lrcc_manager.sent_packets, // UNIQUE: PrivateDoublyLinkedList<sent_packet_fixed>
7      lrcc_manager.sent_packets.repr;  // UNIQUE: set<PrivateNode<sent_packet_fixed>>
8  reads stream_manager, // UNIQUE:StreamManager
9      stream_manager.quic_streams_repr, // UNIQUE: set<quic_stream_mutable>
10     stream_manager.stream_nodes_repr, // UNIQUE: set<PrivateNode<quic_stream_mutable>>
11     stream_manager.stream_lists_repr, // UNIQUE: set<PrivateDoublyLinkedList<
       quic_stream_mutable>>
12     stream_manager.segment_lists_repr, // UNIQUE: set<PrivateDoublyLinkedList<
       qstream_segment_fixed>>
13     stream_manager.segment_nodes_repr; // UNIQUE: set<PrivateNode<
       qstream_segment_fixed>>
14 reads pspace_manager, // UNIQUE: PacketSpaceManager
15     pspace_manager.ps_states_repr, // UNIQUE: set<packet_space_state>
16     pspace_manager.ack_buffers_repr; // UNIQUE: set<buffer<packet_num_t>>
```

Listing 3.2: **Example Frame Specifications**

As one might expect, given the complexity of the specification, the proof obligations can easily overwhelm the SMT solver. For higher-level objects in the hierarchy, the verification time of a single procedure can extend way beyond typical interactive range. The problem is often tied to a massive number of quantifier instantiations originating from the frame specifications and the aliasing reasoning.

3.1.2.1 Workarounds for Frame Reasoning

To counter the scalability issue, we have to employ various tricks and workarounds. However, they all come with their own compromises.

Type-based Separation. We take advantage of Dafny's type-based separation, i.e., to define types that are known to be incomparable. Rather than homogenizing the distinct sub-structures into a single `ghost set<object>`, we maintain ghost representations of each distinct type. (The example above actually reflects this, with comments on the type of each footprint set.) This reduces the SMT's reasoning obligations, since it is "free" that objects of different types are disjoint. However, this approach requires new type definitions, along with additional proof annotations.

Layered Updates. Even with the aforementioned discipline, mutation of a child structure (however deep) requires us to re-establish the invariant of all its parent data structures. Hence we carefully wrap our heap updates in multiple layers: the innermost performs the actual mutation, and the outer layers simply expose these changes to higher levels. This is not ideal, since in many cases, the common thing to do is to perform the

mutation at the highest level, without the boilerplate layers. However, it is a necessary compromise to make proof automation possible at that point.

```
method seq_update(s1:seq<int>, s2:seq<int>)
    returns(s3:seq<int>, s4:seq<int>)
    requires |s1| >= 10 && |s2| >= 10
{
    s3 := s1[5 := 100]; // copy entire seq
    s4 := s2[5 := 200]; // copy entire seq
    assert s3[5] == 100; // PASS: s3 is a new seq
    assert s4[5] == 200; // PASS: s4 is a new seq
    assert s1[5] == 100; // FAIL: update is not in-place
}
```

Listing 3.3: **Sequence Update Example**

Selective Immutability. Another technique we employ is the careful use of immutability. Immutable objects (e.g, datatypes, sequences) are trivial to memory reasoning. While technically they are heap objects, and we pass their references around, the semantics is essentially as if we were passing the values directly, independent of the heap state. For example, let us convert the previous mutable arrays in Lst. 3.1 into using sequences. We no longer need to specify the frame of the procedure, nor how the sequences are aliased, since they are all independent of each other.

However, an immutable structure has the "copy-on-update" semantics upon update, where the original structure is left unmodified. We note how the procedure has to allocate two new sequences, which can be expensive. Therefore, if we indiscriminately apply immutability, we may incur a very high performance cost. We must carefully balance the trade-off so that the verifier can handle the proof load, while the performance does not suffer too much.

Eventually our QUIC$_D$ implementation reached ~10K lines of Dafny. We successfully inter-operate with ourself, as well as other unverified QUIC clients/servers, from the handshake initiation to data transfer, until connection close. QUIC$_D$ adds ~21% throughput overhead compared to the unverified baseline when it comes to transmitting gigabytes of data, which is respectable, but not ideal either.

Considering the significant effort needed to achieve this result, we are not satisfied with the Dafny's memory reasoning capabilities, since our workarounds all involve some form of compromise: type-based separation requires additional proof annotations; layered updates result in extra boilerplate code selective mutability incurs performance penalties. Most detrimentally, these workarounds all *scale poorly* with the size of the system. If the data structure hierarchy grows higher, these workarounds will require even more compromises.

3.1.2.2 Observations on Linearity

While QUIC$_D$ exposes many scalability issues with memory reasoning in APV, it is not without merit. In particular, we notice that the full generality of arbitrary aliasing is often unnecessary. For example, each `Connection` object should be completely independent.

Intuitively, two distinct connections should not share any segments, streams, or any other data structures. Similarly, within a single connection, different streams should not share any segments, and different segments should not overlap. Our example in Lst. 3.2 even contains "`UNIQUE`" in the comments, which we were tracking for our own sanity during the development.

The observation aligns with the notion of *linearity* in type theory [166]. A linearly typed object has a unique "owner" object through which it can be accessed. Therefore, linearly typed objects are logically detached from the rest of the heap, similar to how the immutable objects are. Meanwhile, since linear ownership is exclusive, we can safely update linear objects in place, avoiding the performance penalty that immutable objects would incur (as in Lst. 3.3). More importantly, since the type checker will enforce the unique ownerships, the SMT solver can be relieved from the painful alias reasoning.

While there are obvious advantages, linear types can also be quite restrictive for practical usage. For example, QUIC_D contains a doubly-linked list data structure, where a node can be referenced by its two neighbors, and thus is nonlinear by nature. QUIC_D even has several different usages of the list, (corresponding to the stacked-document-shaped objects in Fig. 3.1,) which unfortunately means that nonlinearity is spread throughout the system.

Therefore, while arbitrary aliasing is the general case in theory, and linearity is the common case in practice, we cannot naively apply linear types to relief our trouble with memory reasoning. Intuitively, our formalism needs to account for nonlinearity when required, while still provide the benefits of linearity when possible. We demonstrate how we may achieve this in Linear Dafny [100].

3.1.3 Untangling the Linearity

In this section, we discuss how we integrate linear types into Dafny. First we highlight the new language features, while expanding on how linearity enables safe and efficient in-place updates on otherwise immutable objects. We then discuss our region-based heap model, where the linear types reduce the memory reasoning burden on the SMT solver, and heap regions retain our ability to express aliasing relations.

3.1.3.1 Ownership Modifiers for Types

In Linear Dafny, we introduce the following ownership modifiers:

- `linear` marks a variable as linear; i.e., it has a unique owner.
- `shared` marks a variable as immutably shared.
- `inout` marks a linear parameter variable as "mutably borrowed" (explained below).

Our ownership modifiers are orthogonal to the ghost vs. non-ghost (unmarked) modifiers. Therefore, we may also have `ghost linear` or `ghost shared` variables, which we defer to

Sec. 3.1.3.2. We first discuss how the ownership modifiers work on the unmarked variables, or *ordinary* variables as we call them.

We limit the modifiers to immutable ordinary variables. Intuitively, linearity does not make sense for value types, e.g., `int` or `bool`. Meanwhile, there is no strong motivation for us to add linearity to mutable ordinary variables, because linearity enables safe in-place updates on immutable objects.

Let us demonstrate how we can perform such update using linear sequences in Lst. 3.4. We cannot duplicate or discard linear variables as we do with ordinary variables. Linear variables are simply "moved" rather than copied upon assignments. For example, on line 8 of Lst. 3.4, we move `l1` to `l1'`, and thus we can no longer access `l1` afterwards. Similarly, on line 10, we give up the ownership of `l1'` as we make the method call, obtaining a "new" linear variable `l3` in return.

```
0  method lseq_set(linear l:seq<int>, i:int, v:int)
1      returns (linear r:seq<int>)
2      // performs in-place update ...
3      // pre/post-condition and body omitted ...
4
5  method lseq_update(linear l1:seq<int>, linear l2:seq<int>)
6      returns(linear l3:seq<int>, linear l4:seq<int>)
7      requires |l1| >= 10 && |l2| >= 10
8  {
9      linear var l1' := l1; // just for demo, move l1 to l1'
10     // l3 := lseq_set(l1, 5, 100); // would be a type error, moved out of l1 already!
11     l3 := lseq_set(l1', 5, 100); // l1' now replaced with l3
12     l4 := lseq_set(l2, 5, 200); // l2 now replaced with l4
13     assert l3[5] == 100; // PASS
14     assert l4[5] == 200; // PASS
15     // assert l2[5] == 200; // also would be a type error, moved out of l2 already!
16 }
```

Listing 3.4: **Linear Sequence Update Example**

On the surface, Lst. 3.4 might look similar to Lst. 3.3, where we used ordinary sequences. If anything, Lst. 3.4 comes with more restrictions from the linear types. However, the more important difference is that linearity has eliminated the possibility where the parameters themselves are aliased, or the parameters have any other live references at all. More concretely, let us consider two strategies to implement `lseq_set`:

(1) We allocate another copy of `l`, mutate the copy, deallocate `l`, and return the copy.
(2) We mutate the `l` in-place and return it.

We note that even though immutable objects are technically on the heap, there is no syntactic construct to express "address of" in the language. That is, we cannot write some code to check if the addresses of `l` and `r` are the same before and after the `lseq_set` call. Therefore, from the perspective of a caller, the two strategies are observationally equivalent. Furthermore, there are no more live references to `l` as we enter `lseq_set`, we can thus safely mutate in-place without any impact on the rest of the heap.

Meanwhile, `lseq_set` is "nice" enough to return an r, so the caller regains the lost ownership through this "new" linear variable. Rather than going through the shenanigan

of pretended lost-and-found, we make the syntax a bit nicer with the `inout` modifier. As we show in Lst. 3.5, the callee can mutably borrow a parameter, borrowing the jargon from Rust. The rule of linearity still applies, so we cannot duplicate or discard a `linear inout` variable. However, we can lend it out to a callee with the right signature, and the type checker will make sure the callee will return it back.

```
method lseq_inout_set(linear inout l:seq<int>, i:int, v:int)
    // performs in-place update ...
    // pre/post-condition and body omitted ...

// this is just a syntactically-sugared version of lseq_update
method lseq_inout_update(linear inout l1:seq<int>, linear inout l2:seq<int>)
    requires |l1| >= 10 && |l2| >= 10
{
    lseq_inout_set(l1, 5, 100); // in-place update
    lseq_inout_set(l2, 5, 200); // in-place update
    assert l1[5] == 100; // PASS
    assert l2[5] == 200; // PASS
}
```

Listing 3.5: **Linear Inout Sequence Update Example**

Shared variables enables read-only aliasing. We can duplicate shared variables, but only under restricted scope. We cannot store shared variables in any data structure, while we can store linear variables in linear datatypes. Therefore, shared variables gives us some flexibility in nonlinear references, but not much more than that.

3.1.3.2 Region-Based Model for the Heap

As we introduce linear types, we also decompose the global heap model in Dafny into *regions*. Our region-based formalism allows us to encapsulate linear data inside nonlinear data and vice-versa, enabling more flexible aliasing relations. More specifically, the API introduces the following definitions:

```
linear datatype Region = Region(Id: RegionId, Allocated: set<Loc>)

function RefLoc<A>(r: RefCell<A>): Loc

function LocRef<A(00)>(l: Loc): RefCell<A>

predicate ValidRef(loc: Loc, id: RegionId, is_linear: bool)

function Get<A>(g: Region, r: RefCell<A>): A

function Modifies(lcs: set<Loc>, g1: Region, g2: Region): (b: bool)

// axioms elided ...
```

Listing 3.6: **Region APIs in Linear Dafny**

We note that `Region` is a ghost linear datatype, which has no runtime impact. Intuitively, a `Region` represents a private heap, keeping track of its allocated objects, so that local changes stay local. We can thus hide a nonlinear data structure inside a linear region,

where externally the data structure has a linear interface, without the need to specify its impact on the global heap via `modifies` clauses.

```
0  method NewRegion()
1      returns (ghost linear g: Region)
2
3  method FreeRegion(ghost linear g: Region)
4
5  function method Read<A>(ghost shared g: Region, r: RefCell<A>): (a: A)
6      requires ValidRef(RefLoc(r), g.Id, false)
7      ensures Get(g, r) == a
8
9  method Write<A>(ghost linear inout g: Region, r: RefCell<A>, a: A)
10   requires ValidRef(RefLoc(r), old(g.Id), false)
11   ensures Get(g, r) == a
12   ensures Modifies({RefLoc(r)}, old(g), g)
13
14 method Alloc<A>(ghost linear inout g: Region, a: A)
15     returns (r: RefCell<A>)
16     ensures ValidRef(RefLoc(r), g.Id, false)
17     ensures Get(g, r) == a
18     ensures Modifies({}, old(g), g)
19     ensures RefLoc(r) ∉ Allocated(old(g))
20     ensures RefLoc(r) in Allocated(g)
```

Listing 3.7: **Region API in Linear Dafny (Cont'd)**

In more detail, we use `RefCell` as the reference type to retrieve objects from a region. We can duplicate and passe `RefCell` around, potentially creating multiple references to the same object. `Modifies` is our per-region equivalent of Dafny's global-heap `modifies` clause. The predicate states that a region stays the same, except for a specified subset or new allocations. The `ValidRef` predicate states that a reference is valid in a given region. The predicate is true *if and only if* the reference came from an `Alloc` (or `AllocLinear`) call with the region, which is how we make sure each region is isolated from the rest of the heap. For example, we would violate the precondition on line 10 in Lst. 3.7, if we attempt to call `Wrtie` with a reference from one region on a different region.

```
0  function method Borrow<A>(ghost shared g: Region, r: RefCell<A>): (shared a: A)
1      requires ValidRef(RefLoc(r), g.Id, true)
2      ensures Get(g, r) == a
3
4  method Swap<A>(ghost linear inout g: Region, r: RefCell<A>, linear inout a: A)
5    requires ValidRef(RefLoc(r), old(g.Id), true)
6    ensures Get(old(g), r) == a
7    ensures Get(g, r) == old_a
8    ensures Modifies({RefLoc(r)}, old(g), g)
9
10 method AllocLinear<A>(ghost linear inout g: Region, linear a: A)
11     returns (r: RefCell<A>)
12     ensures ValidRef(RefLoc(r), g.Id, true)
13     ensures Get(g, r) == a
14     ensures Modifies({}, old(g), g)
15     ensures RefLoc(r) ∉ Allocated(old(g))
16     ensures RefLoc(r) in Allocated(g)
```

Listing 3.8: **Region API in Linear Dafny (Cont'd)**

It is worth noting that memory safety relies on both the SMT solver and the type checker. Specifically, `ValidRef` depends on the region's identifier rather than the specific contents

of the region. As a result, once the SMT solver establishes the validity of a reference in a region, the validity remains invariant, even after the region is freed. Meanwhile, the type checker will make sure that a region is alive whenever we try to use it. For example, after `FreeRegion` consumes a region, we are (syntactically) prohibited from using any references in that region.

So far we have discussed how we can store nonlinear data inside linear structure, but the opposite direction can also be useful. As shown in Lst. 3.8, we can use the `AllocLinear` call to allocate a linear object, (where the linearity is reflected by `ValidRef(...,true)` in the post-condition). Unlike `RefCell` in Rust, we do not need the runtime checks to enforce single mutable borrow exclusive-or multiple immutable borrows. Instead, proof obligations on top of the type system help ensure the safety. Therefore, we can borrow a linear object from a shared region via `Borrow` or swap a linear value in a linear region via `Swap`, without runtime overhead.

3.1.4 Evaluating the Improvement

Now that we have introduced the types and formalism in Linear Dafny, we evaluate how these changes impact our verification experience. Recall our observations in Sec. 3.1.2.2, where a doubly-linked list data structure in QUIC$_D$ stands in the way between us and linearity. Here we first qualitatively demonstrate the new features of Linear Dafny, with a doubly-linked list data structure with a linear interface (Sec. 3.1.4.1). We then perform a large-scale quantitative evaluation on VeriBetrKV$_D$, which is a verified, high-performance storage system, originally developed by Hance et al. [72]. We convert VeriBetrKV$_D$ into using Linear Dafny, and report our findings in terms of proof size and verification time (Sec. 3.1.4.2).

```
0  datatype Option<A> = None | Some(a: A)
1
2  datatype Node<A> = Nil
3      | Cons(data: Option<A>,
4          next: RefCell<Node<A>>,
5          prev: RefCell<Node<A>>)
6
7  type NodePtr<A> = RefCell<Node<A>>
8
9  linear datatype List<A> = List(
10     ghost linear g: Region,
11     sentinel: NodePtr<A>,
12     ghost data: seq<A>,
13     ghost refs: seq<NodePtr<A>>)
```

Listing 3.9: **DLL Datatypes Under a Linear Region**

3.1.4.1 Qualitive Evaluation

In Lst. 3.9, we first show the data structure definitions for the new doubly-linked list. Specifically, the `linear List<A>` datatype contains a ghost linear region `g`, which keeps

track of the private heap "owned" by the list.

We can also specify the data structure invariant, similar to how we would do with the dynamic frames. Note that we have to explicitly specify the aliasing relations, which is necessary due to the nonlinear nature of the list. However, the references are now all `RefCell`, which are scoped to the local region `g`, rather than the global heap.

```
predicate ListInv<A>(list: List<A>)
{
    var g := list.g;
    var data := list.data;
    var refs := list.refs;
    && |refs| == |data| + 1
    && list.sentinel == refs[0]
    && (∀ i :: 0 <= i < |refs| ⟹ RefLoc(refs[i]) in g.Allocated)
    && (∀ i :: 0 <= i < |refs| ⟹ ValidRef(RefLoc(refs[i]), g.Id, false))
    && (∀ i, j :: 0 <= i < j < |refs| ⟹ refs[i] != refs[j])
    && (∀ i :: 0 <= i < |refs| ⟹ Get(g, refs[i]).Cons?)
    && Get(g, refs[0]).prev == refs[|data|]
    && Get(g, refs[|data|]).next == refs[0]
    && (∀ i :: 1 <= i < |refs| ⟹ Get(g, refs[i]).data == Some(data[i - 1]))
    && (∀ i :: 0 <= i < |refs| - 1 ⟹ Get(g, refs[i]).next == refs[i + 1])
    && (∀ i :: 1 <= i < |refs| ⟹ Get(g, refs[i]).prev == refs[i - 1])
}
```

Listing 3.10: **DLL Invariant Under a Linear Region**

Finally, we show some client code that uses the doubly-linked list. We note that it does not require any `modifies` clause. The client does not even need to be aware of the linear region underneath the data structure.

```
method TestList() {
    linear var list := NewList();
    var ptr1 := InsertLast(inout list, 1);
    Remove(inout list, ptr1, 0);
    FreeList(list);
}
```

Listing 3.11: **Client Code using DLL**

3.1.4.2 Quantitative Evaluation

We now turn to a large-scale evaluation using the VeriBetrKV$_D$ key-value store. Similar to QUIC$_D$, VeriBetrKV$_D$ also resorts to workarounds for complex memory reasoning. Unlike QUIC$_D$ however, VeriBetrKV$_D$ is not only verified for memory safety but also for functional correctness, i.e., it behaves like a dictionary data structure. Moreover, VeriBetrKV$_D$ proves crash safety, which means that it guarantees that the system will not lose data or corrupt its internal state in the event of a crash.

We convert VeriBetrKV$_D$ into using Linear Dafny. The original VeriBetrKV$_D$ is written in ~44 kLoc of Dafny, producing ~31 kLoc of C++. We note that our conversion only applies to the implementation layer (with ~24 kLoc). The rest of the codebase (with ~20 kLoc) are dedicated to the higher layers of model and refinement proofs, which do not involve

memory reasoning. Overall, we find that 91% of the implementation layer is amenable to linear memory reasoning, and only 9% requires reasoning about tricky aliasing relations.

In the linearized version, VeriBetrKV$_L$, we find 28% fewer lines of proofs, 30% shorter verification time overall, and among slow methods, the typical verification time is cut nearly in half. It is therefore possible to perform memory reasoning without the compromises made in QUIC. In fact, we achieve faster verification with fewer annotations, and less boilerplate code, while retaining the performance.

We attribute these improvements to the right choice of formalism, informed by observations in practical use cases. Specifically, we recognize the common case of non-aliasing, and we exploit it to alleviate the scalability issues in memory reasoning with SMT solvers.

VeriBetrKV$_D$ and QUIC$_D$ are not the only systems that would benefit from linearity. In particular, a growing body of large, performant systems built in Rust serve as evidence that linear types are practical and effective for ensuring program safety. In followup work, we build Verus [94], a program verifier for Rust that fully takes advantage of the linear types for memory safety, allowing us to focus on other challenging aspects of APV.

3.2 Performing Theory-Specific Reasoning

We continue our discussion on developing proofs, now with Dafny and Verus. We would like to highlight that Verus has a number advanced features leveraging the ownership types in Rust. Most notably, Verus offer strong support for concurrency reasoning, which is notoriously hard. For a detailed discussion on these topics, we refer the reader to Travis's thesis [71]. In this section, we shift the focus from memory reasoning to theory-specific reasoning.

In system software, there is often the need to work with nonlinear integer arithmetic (NIA) and bit-vector (BV). Consider hand-crafted, high-performance cryptographic implementations in assembly (e.g., those in OpenSSL [126]). Intuitively, to prove the code code correct, we need to reason about not only the low-level bitwise operations but also the high-level algebraic functionalities.

In this section, we briefly discuss how we encode NIA and BV proof obligations in Verus, where we based the encoding on existing manual practices in Dafny programs.

3.2.1 Separating Bit-Vector Obligations

While the SMT solver supports the combination of theories, in practice, mixed theories often lead to poor performance and scalability issues. As a result, Dafny users are advised to exert caution when dealing with integers and bit-vectors together [42]. In the past, researchers have also developed dedicated libraries to deal with these theories separately, including in Komodo [59], Ironclad [79], IronFleet [78], and Armada [104].

```
// Dafny can only handle the following with small bit-widths
lemma dafny_right_shift_div(x: uint8) {
  assert ((x as bv8) >> 3) as nat == x / 8;
  // in the SMT query,
  // the casts correspond to int2bv and bv2int
}
```

Listing 3.12: **Dafny BV Example**

However, the manual separation is still insufficient. Consider the following Dafny example, where we prove that right bit-shifting a variable of type `uint16` by 3 is equivalent to diving it by 8. In Dafny, we have to cast the variable back and forth. This also results in type casts in the SMT query, which means we have not avoided the mixed theory problem after all. In fact, the following example would stop working with a variable of type `uint16`.

In Verus, we assist BV reasoning with a special `bit_vector` proof mode. Externally, this mode operates over integers (e.g., in type `u32`). Internally, it translates the proof obligations into a pure BV query without mixing theories. This results in less manual casting and better solver performance.

```
// Verus can handle much larger bit-widths without a problem
proof fn verus_right_shift_div(x: u128) {
    assert(x >> 3 == x / 8) by (bit_vector);
    // in the SMT query,
    // no cast needed, and no mixed theory!
}
```

Listing 3.13: **Verus BV Example**

3.2.2 Discharging Algebraic Properties

We now discuss the algebra solver backend in Verus, which is inspired by existing proof practices in Dafny. Due to the undecidability of NIA, SMT solvers often struggle to discharge NIA obligations efficiently. As a result, it is common practice to disable NIA in the SMT solver (e.g., via the setting `arith.nl=false` in Z3), relying instead on a first-order axiomatization of arithmetic operations. This approach, however, often necessitates extensive manual proof annotations to guide the solver through each step of the reasoning process.

To illustrate the difficulties, we have chosen a rather complex example in Lst. 3.14. For some context, this is taken from a proof of correctness for RSA in Dafny, where `IsModEquivalent` is a wrapper for the `%` operator. It is not necessary to understand all the details, but overall, we are using the `calc` primitive to prove the congruence relation on line 29. The proof is structured as a chain of implications, where each step involves a handful of equational rewrites based on algebraic properties.

```
0 calc ==> {
1     IsModEquivalent(a, Pow(sig, Pow(2, i)) * rsa.R, rsa.M);
```

```
    {
        LemmaMulModNoop(a, a, rsa.M);
        LemmaMulModNoop(Pow(sig, Pow(2, i)) * rsa.R,
            Pow(sig, Pow(2, i)) * rsa.R, rsa.M);
        LemmaMulProperties();
    }
    IsModEquivalent(a * a,
        Pow(sig, Pow(2, i)) * rsa.R * Pow(sig, Pow(2, i)) * rsa.R, rsa.M);
    { LemmaMulIsAssociativeAuto(); }
    IsModEquivalent(a * a,
        Pow(sig, Pow(2, i)) * Pow(sig, Pow(2, i)) * rsa.R * rsa.R, rsa.M);
    { LemmaPowAddsAuto(); }
    IsModEquivalent(a * a,
        Pow(sig, Pow(2, i) + Pow(2, i)) * rsa.R * rsa.R, rsa.M);
    { reveal Pow(); }
    IsModEquivalent(a * a, next_goal * rsa.R, rsa.M);
    { LemmaModMulEquivalentAuto(); }
    IsModEquivalent(a * a * rsa.R_INV,
        next_goal * rsa.R * rsa.R_INV, rsa.M);
    {
        assert IsModEquivalent(next_a, a * a * rsa.R_INV, rsa.M) by {
            montmul_inv_lemma_1(next_a_view, a_view, a_view, rsa);
        }
        LemmaMulIsAssociativeAuto();
    }
    IsModEquivalent(next_a, next_goal * rsa.R_INV * rsa.R, rsa.M);
    { r_r_inv_cancel_lemma(next_a, next_goal, rsa); }
    IsModEquivalent(next_a, next_goal, rsa.M);
}
```

Listing 3.14: **Calc Chain in Dafny**

More generally, we capture this form of algebraic-rewriting proofs with a new feature `gbassert` in Dafny (and its equivalent in Verus is `integer_ring`). In `gbassert` we take an equation as the goal, and a set of supporting equations as the context. We then employ the Singular algebra solver [66], which decides whether the goal is entailed by the context. Now we can simplify the proof to Lst. 3.15, where we can eliminate a large number of intermediary steps.

```
gbassert IsModEquivalent(next_a, next_goal, rsa.M) by {
    assert IsModEquivalent(rsa.R_INV * rsa.R, 1, rsa.M);
    assert IsModEquivalent(a, exp * rsa.R, rsa.M);
    assert IsModEquivalent(next_a * rsa.R, a * a, rsa.M);
    assert next_goal == exp * exp * rsa.R by {
        LemmaPowAdds(sig,  Pow(2, i), Pow(2, i));
        assert exp * exp == Pow(sig, Pow(2, i) * 2);
        reveal Pow();
    }
}
```

Listing 3.15: **Gbassert in Dafny**

We now briefly summarize the mathematical underpinnings in `gbassert`, which is based on polynomial entailment from Prior work [76, 158].

Polynomial Ring Ideals. Fix a set of variables $x_1, ..., x_n$. Let $R = \mathbb{Z}[x_1, ..., x_n]$ be a polynomial ring over the integers; i.e., the elements of R are polynomials with indeterminate

$x_1, ..., x_n$ and integer coefficients. A set $I \subset R$ is an ideal in R iff

$$a + b \in I \quad \forall a, b \in I$$
$$a \times c \in I \quad \forall a, b \in I, c \in R$$

Let $B = \{b_1, .., b_m\} \subseteq R$. B is a generating set of an ideal I, or $I = \text{ideal}(B)$ if

$$I = \left\{ \sum_{i=1}^{m} r_i \times b_i \mid r_1, ..., r_m \in R \right\}$$

The ideal membership problem decides if a polynomial $p \in R$ belongs to an ideal $I = \text{ideal}(B)$. If $p \in I$, then there exists polynomials $r_1, ..., r_m \in R$ such that

$$p = \sum_{i=1}^{m} r_i \times b_i$$

Our Encoding. `gbassert` takes in one goal statement and an arbitrary number of supporting statements as inputs. Each statement must be a congruence, or a direct equality. For example, a congruence statement is in the form of $a_i = b_i (\text{mod } m_i)$, where $a_i,\ b_i,\ m_i$ are integer-typed expressions.

We start with a pass over the input statements to collect existing variables. For invocations of uninterpreted functions, we introduce fresh variables. For example, the source-level function call `msb(x)` might be assigned a variable name t_0, so that occurrences of `msb(x)` will be replaced with t_0.

We then use all of the existing and new variables to construct elements of a polynomial ring R over the integers. The goal statement (at index 0) is in the form of $a_0 = b_0 (\text{mod } m_0)$ and is translated to the polynomial $a_0 - b_0$. Each supporting statement $a_i = b_i (\text{mod } m_i)$ is translated into the polynomial $a_i - b_i + p_i m_i$, where p_i is a freshly introduced variable. Because the supporting statement is proven (by Dafny), forall a_i, b_i, there exists some p_i such that $a_i - b_i + p_i m_i = 0$.

We form a generating set for an ideal B by collecting all of the polynomials from the supporting statements along with the modulus m_0 from the goal statement:

$$B = \{m_0, a_1 - b_1 + p_1 m_1, \ldots, a_k - b_k + p_k m_k\}$$

We then invoke Singular [66] to decide whether the goal polynomial $a_0 - b_0$ belongs to the ideal generated by B. If so, then we conclude that $a_0 = b_0 (\text{mod } m_0)$. The intuition is that if $a_0 - b_0 \in \text{ideal}(B)$, then there exists polynomials $r_0, ..., r_m \in R$

$$a_0 - b_0 = m_0 \times r_0 + \sum_{i=1}^{k} r_i \times (a_i - b_i + p_i m_i)$$

The p_i values discussed above ensure that the summation on the right evaluates to zero. Hence membership effectively shows that there exists $r_0 \in R$ such that $a_0 - b_0 = m_0 \times r_0$, proving that our original goal congruence holds (i.e., that $a_0 = b_0 \text{ mod } m_0$).

3.3 Work Status and Personal Contribution

The work on QUIC (Sec. 3.1.1) was published in S&P'21 [49]. I implemented and proved the memory safety of most of the protocol logic, under the guidance of Jay Bosamiya, and my advisor Bryan Parno. My contribution to the paper writing was scoped to the discussion on the protocol logic. I did not participate in implementing the record (cryptographic) layer, which I have omitted here.

The work on Linear Dafny (Sec. 3.1.3) was published in OOPSLA'22 [100], where we also received a distinguished paper award. I performed most of the conversion (linearization) from the original VeriBetrKV$_D$ code base, with helpful tips from Jialin Li, Andrea Lattuada, and Jon Howell. I also worked on experiments to measure the performance improvement in SMT. My contribution to the paper writing was on the minor side. Chris Hawblitzel was the main contributor to the type system formalization and its implementation, in which I did not participate.

I did the encoding for BV and NIA discussed above as a (small) part of the Verus project, which was published in OOPSLA'23 [94]. I implemented a predecessor of `integer_ring` mode in Dafny for the Galápagos work published in CCS'23 [174].

Chapter 4

Debugging Proofs

So far we have been discussing how to achieve more scalable APV with improved reasoning capability. We have only alluded to automation failure, which is in fact the fundamental concern in APV. When automation fails, the programer needs to step in and debug the proof, stalling further development.

Unfortunately, it can be hard to understand why verification failed or how to fix it. The SMT solver is roughly a black box that gives a binary output (i.e., `unsat` or not). Consequently, a developer also receives a binary verification result (i.e., success or failure), with little to no visibility into the solver's internal state.

Exposing such internal state in a useful manner is challenging, since an SMT solver might prove millions of clauses during its satisfiability check. Hence, it can be difficult to answer questions like "How much progress has the solver made towards the verification goal?" or "What additional facts are needed for the solver to complete the proof?". Indeed, because verification is generally undecidable, the developer does not even know initially if the failure is due to an incorrect proof or incomplete automation.

As a result, when writing verified program in APV, a significant amount of time goes into proof debugging. More specifically, the developer needs to come up with *diagnostic assertions*, which are proof annotations (described in Sec. 2.3.1) to help them better understand the proof failure. In this section, we only refer to source-level assertions (not to be confused with SMT-level assertions).

4.1 Probing the Solver State

Assertion-based debugging has been the de facto practice in APV. We refer the reader to, for example, Dafny's manual assertion guide [44], F*'s guide [63], and various proof debugging questions on StackOverflow [150, 151]. However, while it is a common practice, there is no clear agreement on how to write them, and more importantly, all the steps are

performed manually.

Effective assertion choice and placement is a key part of the proof engineering process. Choosing the wrong assertion or inserting it in the wrong place weds little light on the cause of the proof failure. Further, a single assertion would rarely be sufficient; instead, multiple iterations are often needed to further break down the proof goal, until either the proof succeeds, or the developer determines the key missing facts the prover needs (or finds a bug in the code).

Assertion-based debugging is an arcane art. Beginners find it hard to understand what assertions to add, where to add them, and how to use them to break down the proof goal. We frequently see beginners become stuck randomly adding assertions that do not improve their understanding of the proof failure. Even for experts, assertion-based debugging is tedious and error prone.

```
1  // helper lemma, already proven
2  proof fn mul_inequality(x:int, y:int, z:int)
3      requires x <= y && 0 < z
4      ensures  x * z <= y * z
5  {...}
6
7  // first attempt for bounded integers
8  proof fn mul_inequality_bounded(
9    x: int, xbound: int, y: int, ybound: int
10 )
11   requires 0 <= x < xbound
12   requires 0 <= y < ybound
13   ensures x * y <= (xbound - 1) * (ybound - 1)
14 {
15   // precondition fails for both calls
16   mul_inequality(x, xbound-1, y);
17   mul_inequality(y, ybound-1, xbound-1);
18 }
```

```
1  // second attempt for bounded integers
2  proof fn mul_inequality_bounded(
3    x: int, xbound: int, y: int, ybound: int
4  )
5    requires 0 <= x < xbound
6    requires 0 <= y < ybound
7    ensures x * y <= (xbound - 1) * (ybound - 1)
8  {
9      // Step #2: split the assertion into pieces
10     assert(x <= xbound - 1); // PASS
11     assert(0 < y); // FALL
12     // Step #1: inline precondition
13     assert(x <= xbound -1 && 0 < y); // FALL
14     mul_inequality(x, xbound-1, y);
15     mul_inequality(y, ybound-1, xbound-1);
16 }
```

Listing 4.1: **Verus Expand Definition Example**

We illustrate the challenges with two simplified examples. In real verification projects, the properties involved are much larger and more complex, making the manual manipulation of source-level assertions a remarkably laborious, error-prone process.

We first give an example where a function's precondition fails. After drafting the proof on the left in Lst. 4.1, we learn that the precondition on line 3 fails at both callsites (lines 16 and 17). On the right side of Lst. 4.1, we first copy over the failing precondition, substituting the arguments with the values at the call site, as in line 13. When we run the verifier again, the added assertion fails as expected. To learn which part of the conjunction is the problem, we split the assertion into two separate ones (lines 10 and 11). After running the verifier yet again, we finally identify the problem is that `0 < y` simply does not hold.

At this point, it becomes clear that we are attempting to invoke the lemma in the wrong way, rather than the solver giving up due to incompleteness. However, this is not so obvious initially — we have to go through multiple edit-and-rerun cycles to find out.

In the second example, we examine a case where the post-condition fails. The first step is typically to copy the failing post-condition to the end of the function, so that we can manipulate it further. When that assertion fails (as expected), it remains unknown which of the four branches is causing the failure. We then move the assertion into each branch, as shown on the right side. Now we re-run verification, and observe that the assertion in the third branch fails, where we can start the actual fix.

```
spec fn fibo(n: nat) -> nat {
  if n == 0 { 0 } else if n == 1 { 1 }
  else { fibo(n - 2) + fibo(n - 1) }
}

// initial debug attempt
proof fn fibo_is_monotonic(i: nat, j: nat)
  requires i <= j
  ensures fibo(i) <= fibo(j) // FAIL
{
  if i < 2 && j < 2 {}
  else if i == j {}
  else if i == j - 1 {
    fibo_is_monotonic(i, j - 1);
  } else {
    fibo_is_monotonic(i, j - 1);
    fibo_is_monotonic(i, j - 2);
  }
  assert(fibo(i) <= fibo(j)); // FAIL
}
```

```
// second debug attempt
proof fn fibo_is_monotonic(i: nat, j: nat)
  requires i <= j,
  ensures fibo(i) <= fibo(j),
{
  if i < 2 && j < 2 {
    assert(fibo(i) <= fibo(j)); // PASS
  } else if i == j {
    assert(fibo(i) <= fibo(j)); // PASS
  } else if i == j - 1 {
    fibo_is_monotonic(i, j - 1);
    assert(fibo(i) <= fibo(j));  // FAIL
  } else {
    fibo_is_monotonic(i, j - 1);
    fibo_is_monotonic(i, j - 2);
    assert(fibo(i) <= fibo(j)); // PASS
  }
}
```

Listing 4.2: **Verus Localize Error Example**

Up to this point, our steps have been mechanical and repetitive. Arguably, the actual "interesting" part of the debugging process has just begun. In this case, maybe the inductive hypothesis is not strong enough, or maybe the recursive call arguments need to be adjusted, which are questions that are more worthy of a developer's time and effort. Nevertheless, we have to suffer through the tedious steps first.

4.2 Debugging through ProofPlumber

Since much of the effort of proof debugging goes into automatable operations, we propose proof actions[1] to automatically transform the source program in APV. With proof actions, we can capture in an programmatic manner the existing proof-debugging practices. Not only do proof actions reduce the tedium and transcription errors, they also make it possible to hand off the "wisdom" behind these practices to new developers.

Meanwhile, we also recognize the need to support more than a fixed set of proof actions. Many APV languages are still at a nascent stage, and the proof debugging techniques may evolve as the language matures. We need a way to allow developers to customize their own proof actions, and maybe compose them with others proof actions as well.

[1]This name is inspired by the code actions supported by the Language Server Protocol (LSP)

We thus introduce ProofPlumber, a novel and extensible debugger framework for Verus. ProofPlumber enables easy creation and application of proof actions. Fig. 4.1 illustrates the workflow: ProofPlumber parses and type-checks the source program text from the editor. ProofPlumber then makes the parsed program available to proof actions, which are modular plugins. After a proof action is done manipulating the program, ProofPlumber pretty-printers the modified program back into the editor.

At a high level, the ProofPlumber APIs allows a proof action to do the following:

(1) Lookup context information, e.g., types and definitions.
(2) Manipulate the source-level program and proof.
(3) Interact with the verifier, e.g., run the verifier and get the failed assertions.

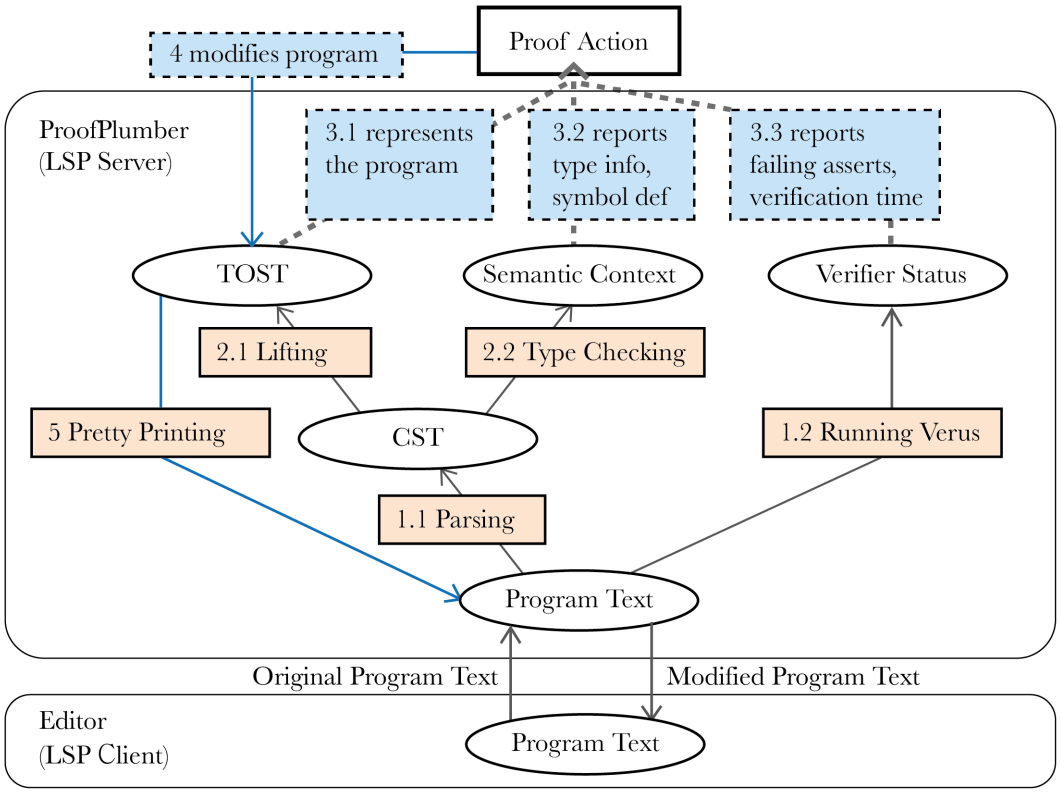


Figure 4.1: **ProofPlumber Workflow**

Fundamentally, a proof action is a procedure that edits the user's source program based on results from type checking and verification. The corresponding data structures in ProofPlumber are: **(a)** Transformation-Oriented Syntax Tree (TOST) nodes representing the user's source code; **(b)** the Context, which contains additional source-level information such as types and definitions; and **(c)** the Verus verifier, which contains information about failing assertions.

TOST is the core data structure that represents the source program. It is an abstract syntax

tree, with each language construct represented as an `enum` (e.g., `assertExpr, blockExpr`). Since the TOST is not a concrete syntax tree (CST), it omits semantically irrelevant syntactic details. The TOST thus allows easy manipulation of the user's source code, ignoring trivialities like whitespace. The TOST offers the following APIs (corresponding to 3.1 represents the program, 4. modifies program, and 5. Pretty printing in Fig. 4.1).

- *Traverse/Edit.* A TOST node allows direct access to its children. For example, `assertExpr` has a field `expr` that contains the asserted expression, which can be accessed and modified directly. Additionally, ProofPlumber offers a visitor pattern for recursively filtering or transforming TOST nodes.
- *Create.* When developing proof actions, it is often necessary to create new TOST nodes. While this can be done through each node's constructor, it can be tedious for large expressions; e.g., consider the expression "`x + y * 4`", which needs five constructor calls. To simplify this process, ProofPlumber provides an option to parse user-provided text into a TOST node.
- *Concretize.* When the proof action is done modifying the TOST, it needs to apply the changes to the program. Since the TOST is abstract, ProofPlumber provides an API to convert it back to a CST, hiding the details of the conversion.

Context is another core data structure for writing a proof action. It contains the following information not easily accessible from the TOST (corresponding to *3.2 reports type info and symbol def* in Fig. 4.1):

- *Node in Scope.* A proof action generally acts within a specific scope indicated by the user; e.g., the user's cursor location may identify an expression that is inside a function, a file, a module, and a crate.
- *Type.* Needless to say, type information is crucial for understanding and manipulating the source program. Rust does not require type annotations for every variable or expression, so the type information is often unavailable at the source (or TOST) level. However, the type of every expression has been computed by the type checker, and this API provides that information.
- *Definition.* It is often necessary to look up the full definition for an identifier. For example, when case matching on an enum variable `c:Color`, it is necessary to look up the variants of `Color`. At the TOST level, the definition of `Color` may be in a different module or even a different crate than the occurrence of `c`. This API provides the definition of an identifier, which can be a name for a struct, an enum, a function, etc.

Verifier (Verus driver) is the last core data structure. It allows the proof action to interact with the verifier. A proof action does not have to finish all of its rewriting in one pass; instead, it can make a change, invoke the verifier, and then continue rewriting based on the verifier's response. Corresponding to *3.3 reports failing asserts and verification time* in Fig. 4.1, this structure provides:

- *Errors.* The list of failing assertions, preconditions, and postconditions.
- *Time.* It often helps to know how long verification takes, since proofs with shorter

verification time are often more robust [173]. If a proof takes too long, the proof action may choose a more efficient one.

Notes on Implementation. As Verus is based on Rust, ProofPlumber extends rust-analyzer [145], the official language server for Rust, to process Verus programs. As with rust-analyzer, ProofPlumber adheres to the Language Server Protocol (LSP) [112]. In turn, proof actions developed with ProofPlumber are compatible with editors that implement the client-side of the LSP.

In our implementation, we have extended rust-analyzer's grammar and parser to obtain the Verus CST (*1.1 Parsing* in Fig. 4.1). We then lift the CST to a TOST, eliminating details like whitespace (*2.1 Lifting* in Fig. 4.1). We construct our Context APIs by extending rust-analyzer's type checking implementation (*2.2 Type Checking* in Fig. 4.1). After a proof action manipulates a TOST node using ProofPlumber APIs, our pretty printer restores the TOST to a concrete program (*5 Pretty Printing* in Fig. 4.1).

4.3 Automating Manual Practices

To demonstrate ProofPlumber's expressivity, and to capture the most common practices in proof debugging, we implemented 17 proof actions. We come up with two groups of proof actions: **(1)** proof actions inspired by Dafny's documentation [44], which suggests a set of manual rewrites for debugging proofs, and **(2)** proof actions distilled from the experiences of Verus developers. We note there are overlaps between the two groups, as APV languages often share similar proof debugging practices.

Dafny provides 29 suggested rewrites for manual proof debugging [44], of which we have implemented 16 as proof actions. We exclude the other 13 rewrites, as 9 of them are not applicable to Verus, and 4 are inherently manual. We implemented 7 additional proof actions to automate routine proof engineering tasks in Verus.

Insert Failing Postconditions. A common proof failure is that a procedure's postconditions cannot be established. Since there can be multiple postconditions and multiple exit points (e.g., due to an early `return`), developers often employ a tedious manual process to pinpoint the failing conditions at each exit point. This proof action automatically adds the failing postcondition(s) at each exit.

Insert Failing Preconditions. When preconditions cannot be established at a call site, this proof action inlines the precondition in the caller's context.

Introduce Match Case Assertions. A special case of assertion decomposition is when the assertion is about an enum. Today, the developer tediously writes a match statement for the enum and then adds assertions to each case to identify where the problem lies. This proof action emits a boilerplate match statement for the enum, but only presents the failing variants.

Remove Redundant Assertions. During proof debugging, to understand the solver's

	Proof Actions	Group	#lines	Verif. Errors	Verif. Time	TOST R/W	TOST Conc.	TOST Create	Lookup Node	Lookup Type	Lookup Defn.
1	Insert Failing Postconditions	Verus	64	✓		✓	✓		✓		
2	Insert Failing Preconditions	Verus	41	✓		✓	✓		✓	✓	✓
3	Introduce Matching Assertions	Verus	84	✓		✓	✓	✓	✓	✓	✓
4	Remove Redundant Assertions	Verus	56	✓	✓	✓	✓		✓		
5	Apply Induction	Verus	125	✓	✓	✓	✓	✓	✓	✓	✓
6	Weakest Precondition Step	Verus/Dafny(5)	176			✓	✓		✓	✓	✓
7	Insert Assert By	Verus/Dafny	29			✓	✓		✓		
8	Decompose Failing Assertion	Verus/Dafny	100	✓		✓	✓		✓	✓	✓
9	Split Implication in Ensures	Dafny	48			✓	✓		✓		
10	Split Smaller or Equal to	Dafny	71			✓	✓		✓		
11	Convert Implication into If	Dafny	40			✓	✓		✓		
12	Introduce Assert Forall	Dafny	42			✓	✓		✓		
13	Introduce Assert Forall Implies	Dafny	57			✓	✓		✓		
14	Introduce Assume False	Dafny	30			✓	✓	✓	✓		
15	Reveal Opaque Function Above	Dafny	45			✓	✓		✓		✓
16	Reveal Opaque Function in Block	Dafny	48			✓	✓		✓		✓
17	Add Seq "in-bounds" Predicate	Dafny	58			✓	✓		✓	✓	✓

Table 4.1: **Proof Actions Implemented with ProofPlumber**

state, proof engineers typically introduce multiple assertions, most of which are redundant (i.e., they help the human, not the verification). Hence, after debugging a proof, to maintain source code readability, developers manually remove these redundant assertions. This proof action mechanizes the process.

Apply Induction. If the selected variable is a natural number or an abstract datatype, this proof action generates the boilerplate code for an inductive proof, including the base and inductive cases. When the selected variable is an enum, it introduces a match statement with an empty proof block for each variant, generating the recursive call to the lemma when the variant is defined recursively.

Weakest Precondition Step. This proof action moves an assertion above the statement that precedes it: in the case of a branch statement, it moves the assertion to the end of each of the branch statements. More generally, the proof action implements the rules of the weakest precondition calculus.

Decompose Failing Assertion. When a complex assertion fails to verify, it is not always obvious which portion of the failing expression is responsible. This proof action automates the process of decomposing and isolating the failing sub-formulas.

As a qualitative illustration on one can implement a proof action using ProofPlumber, we show a snippet for *Decompose Failing Assertion* in Lst. 4.3. This proof action uses all three of ProofPlumber's APIs to analyze a failing assertion with a conjunction of clauses

and present the clauses that fail. Specifically, the proof action retrieves the surrounding function using the **Context** API (line 6). Inside the function, the original assertion is replaced (using the **TOST** API) with an assertion of one conjunct (line 11). The proof action then uses the **Verifier** API to invoke the verifier on this modified function (line 12) and check if the new assertion fails. If so, it is added to the source code.

```
1  fn decompose_failing_assertion(
2      api: &AssistContext<'_>,  // handle for API calls
3      assertion: AssertExpr,      // TOST Node to modify
4  ) -> Option<BlockExpr> {      // ``None'' indicates the proof action is not applicable
5      let split_exprs = split_expr(&assertion.expr)?;  // split into logical conjuncts
6      let this_fn = api.tost_node_in_scope::<Fn>()?; // Find assertion's enclosing ``Fn''
7      let mut stmts: StmtList = StmtList::new();
8      for e in split_exprs {
9          // make each logical conjunct into an assertion of its own
10         let split_assert = AssertExpr::new(e);
11         let modified_fn = api.replace_statement(&this_fn, assertion, split_assert)?;
12         if api.run_verus(&modified_fn)?.is_failing(&split_assert) {
13             stmts.statements.push(split_assert.into());
14         }
15     }
16     stmts.statements.push(assertion.into());  // restore the original assertion
17     Some(BlockExpr::new(stmts))
18 }
```

Listing 4.3: **Implement Decompose Failing Assertion**

In Tab. 4.1, we have summarized the proof actions we implemented and their API usages in ProofPlumber. Generally, we can capture and automate many of the mechanical steps in the proof debugging process, in a fairly succinct and expressive manner (29~176 lines code of for each action).

4.4 Work Status and Personal Contribution

The work on ProofPlumber was published at CAV'24, where we also received a distinguished paper award. Please note that the lead author, Chanhee Cho, implemented the majority of ProofPlumber itself and most of the proof actions. Nevertheless, ProofPlumber was heavily influenced by my own experience with debugging proofs in Dafny. At the start of the project, I came up with a set of proof actions, which eventually lead to Tab. 4.1. I also implemented several proof actions with and without the framework, which shaped the design of ProofPlumber APIs. I made minor contributions to the implementation of ProofPlumber itself (e.g., in constructing the TOST), and I made moderate contributions to the paper writing.

Chapter 5

Reusing Proofs

Proof engineering, while not a commonly used term, can be seen as a natural parallel to software engineering. Although software engineering typically focuses on the design, creation, and maintenance of conventional software, we argue that the core principles such as abstraction, modularity, and reuse, are equally relevant in managing complexity and achieving scalability in verified software.

In this chapter, we demonstrate these principles in the context of verified low-level cryptography (Sec. 5.1). Specifically, we are interested in scaling up the support for multiple heterogeneous hardware platforms. For instance, if we have verified an ECDSA [83] implementation on x86, we would ideally not have to start from scratch for another implementation on ARM. The key question is how to define proper abstractions, so that we can leverage the commonalities between different implementations, without sacrificing the performance benefits of architecture-specific optimizations.

To address these challenges, we introduce Galápagos, a framework designed to efficiently verify low-level cryptography across multiple platforms. As a more general contribution, Galápagos includes a standard library for Dafny (Sec. 5.2.1), which consolidates and enhances libraries from prior projects. This library, now actively maintained and distributed by AWS, demonstrates how proof engineering can transform repetitive efforts into reusable, standardized resources.

Building on this foundation, we leverage ML-style functors [113] to create reusable abstractions (Sec. 5.2.2), enabling modular and scalable verification. We then show how Galápagos reduces redundancy in hardware ISA specifications (Sec. 5.2.3) and facilitates the reuse of algorithmic reasoning across implementations (Sec. 5.2.4). Finally, we evaluate Galápagos on real-world use cases, demonstrating significant reductions in verification effort and proof burden across multiple heterogeneous hardware platforms (Sec. 5.3).

5.1 Verifying Low-level Cryptography

Cryptographic primitives are often on a system's critical path, making high performance a crucial requirement. Historically, cryptographic providers such as OpenSSL have met the performance demands via hand-written assembly code that utilizes platform-specific optimizations (e.g. NEON [7] or AESNI [67]). Emerging heterogeneous platforms reinforce this trend, since compilers may not be available until long after the platforms are deployed, necessitating hand-crafted assembly code.

Unfortunately, manually writing such low-level code invites vulnerabilities; e.g., OpenSSL has reported 33 CVEs between 2021 and 2023 [127], of which 29 are memory safety or function correctness bugs. Formal software verification can statically prove an implementation free of entire classes of vulnerabilities, but prior work in this area is ill suited to a world of heterogeneous hardware, where verification cost and specialization gain are at odds.

A large swath of work [5, 17, 57, 137, 156, 170, 177] verifies high-level source code and then trusts the compiler to produce the correct assembly. This approach reduces verification costs but sacrifices specialization-based performance [23], and it is also infeasible for emerging platforms without a compiler. Another body of work directly targets assembly implementations [2, 3, 23, 25, 35, 138, 140, 158]. This approach retains performance but targets only specific platforms. To support an algorithm (say, ECDSA [83]) on a new platform, the developer needs to start from scratch, which is not-scalable.

5.1.1 Disassembling the Monolithic Approach

Galápagos builds atop of Vale/Dafny toolchain [23], which takes a monolithic approach to assembly code verification. Conceptually, we needs the following components:

(1) **Functional Specification** for the input/output behavior of the primitive.
(2) **ISA Specification** describing the machine model and instruction semantics.
(3) **Assembly Code** implementing the cryptographic routine.
(4) **Refinement Proof** connecting the semantics of (3) over (2) to (1).

Let us begin with the assembly code (3) written in Vale. Vale is a domain-specific language designed for writing and verifying assembly code. It embeds the assembly instructions into a backend verifier, which in our case is Dafny, to discharge the associated proof obligations.

In Lst. 5.1, we show a code snippet written in Vale. The `reads` and `modifies` clauses in the procedure are similar to those in Dafny, but now they are specific to the machine model, which consists of low-level state such as registers and memory. It is worth clarifying that Vale procedures are merely marcos. Therefore, invoking `times4()` in another Vale procedure does not create a stack frame in the underlying machine model. Meanwhile, `RV_ADD` is an assembly instruction, which updates the registers in the machine state.

```
procedure times4()
  requires a0 < 100;
  reads a0;
  modifies a1, a2;
  ensures a2 == 4 * a0;
{
  RV_ADD(a1, a0, a0);  // a1 <- a0 + a0
  RV_ADD(a2, a1, a1);  // a2 <- a1 + a1
}
```

Listing 5.1: **Sample RISC-V Code in Vale**

We now turn to the ISA specification (2). First we need to define the machine state, which includes a set of 32-bit registers and a flat memory.

```
datatype Reg32 =
  | a0
  | a1
  | ...

datatype state = state(
  regs: map<regs_t, uint32>,  // 32-bit registers
  flat: map<int, uint8>,  // Flat memory
  ok: bool)      // Not crashed

// base integer instruction set, 32-bit
datatype Ins32 =
  | RV_ADD (rd: Reg32, rs1: Reg32, rs2: Reg32)
  | RV_LW (rd: Reg32, rs1: Reg32, imm12: uint32)
  | ...
```

Listing 5.2: **RISC-V Machine Model in Dafny**

We then show the instructions semantics in Lst. 5.3, where `eval_code` defines the semantics as a function[1]. We note that there are special primitives in Dafny/Vale connecting the semantics of `Ins32.RV_ADD` here to the instructions in Lst. 5.1. We have elided the details for brevity.

```
function method eval_ins32(ins: Ins32, s: state): state
{
  match ins
    case RV_LW(rd, rs, imm) ⇒
    // load word from s.flat[rs + imm], set ok to false if unaligned
    ...
}

function method eval_code(c: code, s: state): state
{
  match c
    case Ins32(ins) ⇒   eval_ins32(ins, s)
    case Block(block) ⇒   eval_block(block, s)
    ...
}
```

Listing 5.3: **RISC-V ISA Semantics in Dafny**

[1]We use `function method` here, which makes the semantics executable, and thus allows for fuzz testing the semantics against the actual hardware.

To make the discussion on the rest of the components more concrete, let us consider the RSA-3072 signature verification routine as an example. Recall the four components of the monolithic approach: (1) the functional specification, (2) the ISA specification, (3) the assembly code, and (4) the refinement proof. For RSA, (1) is straightforward: given a signature s and the public key (e, m), the goal is to compute and output $h = s^e \mod m$. While (1) remains unchanged across platforms, (2) and (3) vary by definition. At first glance, it might seem inevitable to conduct (4) in a platform-specific manner, as the monolithic approach offers little incentive to decompose the refinement. However, for our purposes, we can break it down into two additional components:

(5) **Algorithmic Description** bridging the gap between the specification in (1) and the code in (3). While the specification defines *what* to compute, we also need to prescribe *how* to compute it. For instance, we may choose the Montgomery method [114] for modular exponentiation, where Alg. 1 outlines the concrete steps to follow.

(6) **Machine Interpretation** specifying how the machine state corresponds to the variables in (5). For example, we might decide that register `a0` points to a heap buffer containing 384 bytes representing the input s, and that the same buffer should be reused to store the output h.

While the algorithmic description and machine interpretation are often implicit, we leverage them as explicit abstractions to reduce the verification effort in Galápagos.

5.2 Building the Abstractions in Galápagos

Galápagos is a framework designed to verify low-level cryptographic primitives across multiple heterogeneous platforms efficiently. For each ISA, Galápagos provides a proven-correct, higher-level abstraction of the machine semantics (Sec. 5.2.3). For each cryptographic primitive, developers prove an algorithmic description (Sec. 5.2.4) once, enabling its reuse across different platforms. Developers can use the high-level machine interface to write assembly implementations (Sec. 5.2.5), maintaining a close semantic correspondence with the algorithmic description.

To enable these abstractions, Galápagos extends Dafny with verified functors (Sec. 5.2.2) and leverages a comprehensive Dafny standard library (Sec. 5.2.1), streamlining both algorithmic reasoning and platform-specific verification.

5.2.1 Curating a Dafny Standard Library

Dafny provides a basic set of language features (e.g., sequences or maps). However, any additional properties must be proven from scratch by the developer. As a result, previous Dafny projects [23, 33, 59, 73, 78, 79, 104] have each developed their own project-specific libraries. This has contributed to significant duplication of effort across projects and even across time, as these project-specific libraries are typically not maintained as Dafny actively

evolves.

Early in Galápagos development, we observed that we would need many of the same properties proven by previous projects, so rather than adding yet another project-specific collection, we have created the first Dafny standard library. In creating the new library, we drew upon code and proofs from these previous projects, but rewrote them in a uniform style (both syntactically and in proof style). We also extended them to fill in obvious gaps. We discuss the main components of the library below.

Data Structures. Dafny provides built-in support for sequences, maps, and sets, making them convenient for modeling a wide variety of systems. On top of these functional data structures, we added more robust support for performing and reasoning about insertion, removal, extrema, subsequencing or subsetting, conversions between data structures, and higher-order functions (fold, filter, etc.) over the data structures.

Big Integers. As we discussed in Sec. 2.3.3, cryptographic algorithms often operate on large integers that cannot fit into a single machine word. We provide a parameterized library for representing such large integers as multi-limb sequences. The library includes operations such as `big_add` shown in Lst. 5.12, lemmas about results of the operations, and lemmas describing the effect of converting between large integers represented by different bases. The latter simplify the reasoning about, say, converting the representation of a number as a sequence of bits into a sequence of 32-bit words.

	Line Count	Definitions	Lemmas
Data Structures	1,219	46	40
Big Integers	914	27	29
Nonlinear Arith.	3,732	7	249

Table 5.1: **Dafny Standard Library Statistics.**

Non-linear Arithmetic. As discussed earlier, another common theme in cryptographic proofs is algebraic reasoning. While fragments of non-linear reasoning can be decided the problem as a whole is undecidable. SMT solvers rely on various heuristics to nonetheless try to solve at least some non-linear problems. Unfortunately, in our experience (and that of previous work [59, 79]), such heuristics are unreliable; creating instability (which is our topic in Chapter 6). To mitigate these effects, our library proves a set of common algebraic properties from first principles and make them available as lemmas.

We offer varying levels of automation in the non-linear algebraic properties. Users can invoke very general lemmas (e.g., exposing lots of properties about multiplication), which provide significant automation but may create proof performance problems. Alternatively, developers can invoke tailored lemmas that specify one property (e.g., multiplication is commutative) or even choose a version where they specify exactly which variables in an equation the property should be applied to (e.g., they can specify x and y as arguments to the lemma to show that $x * y = y * x$). These more specific versions require more manual developer work but they provide consistently provide fast, deterministic performance.

The library has been adopted by the Dafny team at Amazon, who have added it to Dafny's continuous integration tests, which run on each commit to the main Dafny repository. The presence of a unified standard library has already encouraged additional contributions from other Dafny developers, including support for monadic operations, searches, sorts, and a Unicode library.

5.2.2 Creating Abstractions with Functors

As we mentioned earlier, we base Galápagos on Vale, with Dafny as the backend verifier. However, the standard Dafny module system is not expressive enough to support our need of abstraction. We thus extend Dafny's modules with verified, ML-style functors, while maintaining compatibility. This requires adapting higher-order functional concepts to Dafny's imperative, first-order design.

5.2.2.1 Limitations of Dafny Modules

Akin to a module in many programming languages, a Dafny module is a collection of types, functions, and proofs. We first discuss the limitations of Dafny's existing module system with an example on number theoretic transform (NTT) [136]. Suppose that we wish to perform efficient polynomial multiplication via forward/inverse NTT. Moreover, we also wish to parameterize our implementation generically over any ring.

```
abstract module Ring {
  type elem // unspecified type

  function unit(): elem

  function add(a: elem, b: elem): (c: elem)
    ensures b == unit() ⟹ c == a // specifies idempotency

  // other ring functions/axioms elided
}

module IntRing refines Ring {
  type elem = int

  function unit(): elem { 0 }

  function add(a: elem, b: elem): elem { a + b }  // PASS: maintains idempotency

  // function add(a: elem, b: elem): elem { b - a } // FAIL: violates idempotency
}
```

Listing 5.4: **Defining an Abstract Module in Dafny**

Dafny modules can be *abstract*, meaning they can declare types and functions without providing their implementations. For example, in Lst. 5.4, the abstract module `Ring` declares a type `elem` and functions operating on it.

Concrete modules can refine abstract modules by providing implementations. For instance, the `IntRing` module refines `Ring` by setting `elem` to the concrete type `int` and providing function bodies. Dafny enforces the refinement relation, ensuring that the concrete definitions satisfy the properties specified in the abstract module. For example, the `add` function in `IntRing` must uphold the idempotency property declared in `Ring`.

Dafny also allows an abstract module to `import` other abstract modules, providing access to their contents. Continuing with our example in Lst. 5.5, we now implement forward NTT generically over any ring. In `FNTT`, we can use the syntax `import R: Ring` to use an unspecified module `R` that promises to implement the `Ring` interface. Now we can use functions in `R` to perform more complex operations without assuming a particular implementation of `R.add`. We can also implement inverse NTT generically in a similar way.

```
abstract module FNTT {
  import R: Ring

  function double(a: R.elem): R.elem { R.add(a, a) }
  // other generic implementations elided...
}

abstract module INTT {
  import R: Ring
  // generic implementations elided...
}

abstract module PolyMul {
  import F: FNTT
  import I: INTT
  // cannot express that F.R is the same module as I.R
  function problematic(a: F.R.elem, b: I.R.elem): F.R.elem {
    F.R.add(a, b)   // ERROR: this does not type check
  }
}
```

Listing 5.5: **Interoperating Abstract Modules in Dafny**

However, Dafny's basic module system falls short in a subtle but important case. Suppose that we now implement polynomial multiplication in `PolyMul` module. Naturally, we would like to leverage our previous NTT modules. However, as we show in Lst. 5.5, when we try to interoperate between the two modules, Dafny has no way to specify that `F` and `I` are parameterized by the same underlying ring.

5.2.2.2 Extending Dafny with Verified Functors

We thus extend Dafny with verified functors. Functors are functions from modules to modules. In our implementation, a functor is a module that takes other modules as arguments (each argument is given a type defined by an abstract module), and the code and proofs in the functor are written in terms of the module arguments. The developer can instantiate the functor by applying it to concrete modules that refine the formal arguments' interfaces. A functor thus allows a collection of code and proofs to be reused when instantiated with different module arguments.

Using functors, we can now successfully implement the polynomial multiplication example. As shown in Lst. 5.6, `FNTT` is now a functor that takes a module `R` of type `Ring` as an argument and returns an instantiation of the FNTT code and proofs specific to that concrete argument. Applying `FNTT` to a different ring module produces a different concrete instantiation. The crucial benefit of using functors (as opposed to Dafny's existing module system) is that when two functors are applied to the *same* argument (e.g., the `Ring` module in `PolyMul`), we can unify the types coming from the two different instantiated modules. Below, we expand on our functor design choices using Dreyer's terminology [53].

```
abstract module FNTT(R: Ring) { /* details elided */ }

abstract module INTT(R: Ring) { /* details elided */ }

abstract module PolyMul(R: Ring, F: FNTT(R), I: INTT(R)) {
  // Functions in F and I can interop since R is the same in both
}
```

Listing 5.6: **Using Functor in Galápagos**

Applicative. Our functors are applicative, meaning that applying the same functor to the same argument(s) in two different contexts still produces the same concrete module. This is crucial for unifying types in examples like Lst. 5.6. Our design contrasts with SML's generative functors, where each application generates a fresh copy of types, even with the same argument module(s). For example, in `A = FNTT(IntRing)` and `B = FNTT(IntRing)`, `A.elem` and `B.elem` will not be of the same type with generative functors.

Second-Class, First-Order. Similar to most ML dialects, our functors are second class, meaning the module system exist in a different plane from ordinary functions and types. Specifically, a module cannot be passed to or returned from ordinary functions, nor can it be stored in datatypes. Our functors are close to being first-order, since they cannot be partially applied, but they can be parameterized by other functors, which is a higher-order property.

Proof Obligations. Unlike most other functor-supporting languages such as OCaml or ML, Dafny's types and methods come with verification obligations. Hence, when extending Dafny to support functors, we had to carefully ensure that the proof of a functor's correctness relies only on the properties promised by the abstract module "types" of its formal parameters, not any details of the concrete instantiations. In exchange, we gain verification efficiency: we need only verify the algorithmic description once; i.e., no additional verification work is required when instantiating the functor with concrete module arguments, since those arguments have already been proven to refine the corresponding abstract modules.

The functor support enables us to decouple the concerns of machine-specific implementation from algorithmic correctness in a parameterized way, which we discuss in the next two sections.

5.2.3 Abstracting the Machine Specification

A developer using Galápagos needs to define the ISA semantics of target platforms, as in all other assembly verification projects. Galápagos facilitates this process with an abstract machine model, which generically defines various bit-level operations. Moreover, Galápagos provides a higher-level memory interface, lifting the semantics of low-level memory accesses closer to the algorithm-level reasoning.

Abstract Bit-Vector Operations. Below, we present some of the operations in the abstract machine. We note that the machine has an under-defined type `uint` parameterized by a radix. This allows the developers to instantiate the abstract machine with the specific word size of their target platform. By doing so, developers also gain access to pre-defined bit-level operations, which they can directly utilize in defining the ISA specification, such as the RISC-V machine model shown in Lst. 5.3.

```
0  abstract module machine_generic {
1   // symbolic upper bound on word size
2   // concrete instantiations must satisfy the ensures clauses
3   function RADIX(): (v:nat)
4    ensures (v > 1)
5    ensures (v % 2 == 0)
6
7   // defines an unsigned integer type upper-bounded by RADIX()
8   type uint = i: int | 0 <= i < RADIX()
9   // defines a 1-bit unsigned integer type
10  type uint1 = i: int | 0 <= i < 2
11
12  // the followings are generic operations
13  // obtained "for free" by concrete
14  // instantiations that define RADIX()
15
16  // word-sized addition with carry
17  function addc(x:uint, y:uint, cin:uint1): (uint, uint1) {
18   var sum := x + y + cin;
19   // handle possible overflow
20   var sum' := if sum < RADIX() then sum else sum - RADIX();
21   var cout := if sum  < RADIX() then 0 else 1;
22   (sum', cout)
23  }
24
25  // extract the most-significant bit
26  function msb(x:uint): uint1 {
27   if x >= RADIX()/2 then 1 else 0
28  }
29
30   // more operations elided ...
31 }
```

Listing 5.7: **Abstract Operations in Galápagos**

Structured Memory Model. Galápagos also offers a verified module that translates a machine's byte-level memory into a structured memory model. As shown in Lst. 5.9, `mem_t` contains a heap and a stack, where a heap is a map from base addresses to sequences of words, and a stack is a sequence of frames. We note that the `uint_t` is under-specified, as in the abstract operations.

```
datatype mem_t = mem(
  // The abstract heap is a collection of disjoint buffers,
  // each identified by its base address
  heap: map<nat, seq<uint>>,
  // The stack is a sequence of frames
  // each frame is a pair of a frame pointer and a sequence of uints
  stack: seq<(nat, seq<uint>)>
)

predicate mem_inv(mem: mem_t, flat: map<int, uint8>)
{
  // detail elided ...
}
```

Listing 5.8: **Memory Abstraction in Galápagos**

We also define `mem_inv`, which describes the refinement relation between structured and flat memory. At a high level, `mem_inv` states that each heap buffer or stack frame maps to the contents of a contiguous block of flat memory, starting at the respective base or frame address. The address conversion between byte-arrays and under-specified word-arrays is straightforward and yet tedious. We thus have elided the details here.

```
datatype iter_t = iter_t(
  base: nat, // start of the heap buffer
  index: nat,    // current index
  buff: seq<uint> // abstract view of the heap buffer
)

predicate iter_inv(mem: mem_t, iter:iter_t, addr: int)
{
  && iter.base in mem.heap
  && mem.heap[iter.base] == iter.buff
  && iter.index <= |iter.buff|
  && iter.index >= 0
  && iter.base + iter.index * RADIX() == addr
}
```

Listing 5.9: **Iterator in Galápagos**

Galápagos further provides an iterator interface for heap access with the type `iter_t`, which abstracts over a structured heap entry. The invariant `iter_inv` states that the iterator is in-sync with the heap entry, and reflects the semantics of a given address in the structured memory.

More importantly, we provide a proof that the structured memory refines the flat memory. For each memory operation on `mem_t`, we show that it has a corresponding operation on `flat_t` which would preserve the refinement in `mem_inv`. In the example below, the lemma shows that a write via an iterator corresponds to a `write_word` on the flat memory.

```
lemma write_iter_preserves_inv(
  flat: map<int, uint8>,
  mem: mem_t,
  iter: iter_t,
  addr: int,
  value: uint)
```

```
6
7   requires
8       && iter.index != |iter.buff|
9       && mem_inv(mem, flat)
10      && iter_inv(mem, iter, addr)
11  ensures
12    var flat' := write_word(flat, addr, value);
13    var iter' := iter.buff[iter.index := value];
14    var mem' := mem[iter.base_ptr := iter'.buff];
15      && iter_inv(mem', iter', addr)
16      && mem_inv(mem', flat')
```

Listing 5.10: **Iterator Write Preserves Memory Refinement.**

We note the refinement proof is a one-time effort by Galápagos, where we provide similar lemmas for the following operations:

- `write_iter`/`read_iter`: heap access via an iterator, corresponding to flat memory access at `iter.base + iter.index * RADIX()`.
- `push_frame`/`pop_frame`: manages the stack frames, corresponding to changes to the stack pointer.
- `write_frame`/`read_frame`: interacts with the top stack frame, corresponding to flat memory access using the frame pointer as the base plus some offset.

The developer only needs to wrap the memory-related assembly instructions with the corresponding high-level operations, and invoke the refinement lemmas. For example, in Lst. 5.11, the `RV_LW` instruction is hidden under the `read_iter` procedure, giving it much higher-level semantics.

```
procedure read_iter(dst: reg32, src: reg32, offset: imm12,
  ghost inc: bool, ghost iter: iter_t)
  returns (ghost iter': iter_t)
reads
  src, flat;
writes
  dst;
requires
  iter.index != |iter.buff|; // Not at the end of the buffer
  // mem and flat are global state variables
  mem_inv(mem, flat);
  iter_inv(mem, iter, src + offset);
ensures
  dst == iter.buff[iter.index];
  inc ⟹ iter_inv(mem, iter', src + offset + 4);
  !(inc) ⟹ iter_inv(mem, iter', src + offset);
{
  // actual assembly instruction
  RV_LW(dst, src, offset);
  // invariant lemma, courtesy of galapagos
  read_iter_preserves_inv(flat, mem, iter, src + offset);
  // ghostly update the iterator
  iter' := iter.buff[iter.index := dst];
}
```

Listing 5.11: **Vale Procedure to Read via Iterator**

We note that the developer does not need to worry about the details of the refinement proof, or even the definition of `mem_inv`. They just need have a valid `iter_t`, call the

`read_iter_preserves_inv` lemma, and now `read_iter` can also return a valid `iter_t`.

5.2.4 Abstracting the Algorithmic Reasoning

A developer using Galápagos writes the high-level algorithm (with proofs), also parameterized by the abstract machine. They can use as many named variables as they wish, unconstrained by finite registers; they can interact with immutable sequences of structured data, rather than byte-level memory accesses. In this way, they can focus on proving the algorithm's mathematical correctness, without worrying about low-level details such as register allocation or heap management.

We first show an example of abstract multi-word addition algorithm below. Algorithms like RSA operate over large integers that cannot fit into a single machine word and must instead be represented by a sequence of words. Here we perform addition over word sequences, using `ops.addc` from our abstract machine. We can also prove the correctness of the algorithm generically, for any radix, and then instantiate it for specific platforms.

```
0  abstract module big_add_generic(ops: machine_generic)
1  {
2    type words = seq<ops.uint>
3    // Interpret a sequence of uint as a natural number
4    function to_nat(xs:words): nat {
5      // Actual definition elided...
6    }
7
8    function big_add(xs:words, ys:words, cin:uint1):(words, uint1)
9      requires |xs| == |ys|
10   {
11     var len := |xs|;
12     if len == 0 then
13       ([], cin)
14     else
15       var (zs, cin') := big_add(xs[..len-1], ys[..len-1], cin);
16       var (z, cout) := ops.addc(x[len-1], y[len-1], cin');
17       (zs + [z], cout)
18   }
19
20   lemma big_add_correct(xs:words, ys:words, zs:words, cout:uint1)
21     requires |xs| == |ys|
22     requires (zs, cout) == big_add(xs, ys, 0)
23     ensures |zs| == |xs|
24     ensures to_nat(xs) + to_nat(ys)
25       == to_nat(zs) + cout * pow(BASE(), |xs|)
26   { /* Actual proof elided */ }
27 }
28
29 module test {
30   import big_add_32_bits = big_add_generic(machine_32_bits)
31   // Free to use the 32-bit version of big_add and big_add_correct
32 }
```

Listing 5.12: **Generic Multi-Word Addition**

5.2.5 Proving a Low-Level Implementation

The developer then writes the assembly implementation in Vale. They can do this by transcribing the assembly output by a compiler (e.g., when run on C reference code), by handcrafting the Vale assembly to exploit optimization opportunities missed by a generic compiler, or any mix of these strategies.

As they write their implementation, they interact the the high-level, structured memory interface (Sec. 5.2.3), which makes it straightforward to invoke the definitions and proofs from the hardware-specific instantiation of the algorithmic description (Sec. 5.2.4).

```
procedure big_add(ghost x_iter:iter_t, ghost y_iter:iter_t, ghost z_iter:iter_t)
  returns (ghost z_iter': iter_t)
  modifies
    t1; t2; a1; a2; a3; a4; mem; flat;
  requires
    mem_inv(mem, flat);
    iter_inv(x_iter, mem, a1) && x_iter.index == 0 && |x_iter.buff| == 96;
    iter_inv(y_iter, mem, a2) ...
    iter_inv(z_iter, mem, a3) ...
  ensures
    iter_inv(z_iter', mem, a3);
    mem_inv(mem, flat);
    a4 == 0 || a4 == 1; // Carry out bit
    to_nat(x_iter.buff) + to_nat(y_iter.buff) ==
      to_nat(z_iter'.buff) + a4 * pow(BASE(), 96);
    // elided the relation between old(mem) and mem ...
{
  RV_ADDI(t1, a3, 384);

  // Implementation code here with loops maintaining iter_inv
  while (a3 < t1)
    invariant
      iter_inv(x_iter, mem, a1);
      ...
  {
    // read_iter maintain iter_inv for free
    x_iter = read_iter(t2, a1, 0, true, x_iter);
    // we just need to increment the register accordingly
    ADDI(a1, a1, 4);
    ...
  }
  ...
  // Invoke concretized lemma from the algorithmic description
  big_add_correct(x_iter.buff, y_iter.buff, z_iter'.buff, a4);
}
```

Listing 5.13: **Vale Procedure for Multi-Word Addition**

To illustrate this process, we present an excerpted version of RISC-V implementation of multi-word addition. We note that the iterators are always passed as ghost arguments, which are logically connected to the underlying machine state via our abstraction. The `read_iter` and `write_iter` procedures provide a convenient way to interact with heap buffers while maintaining the `iter_inv` invariant automatically. We note that `read_iter` also allows the developer to specify whether the iterator should be incremented after reading. In this case, since we set the `ghost inc` flag to true, we also update the register accordingly.

Additionally, iterators enable reasoning about heap buffers as high-level sequences of

structured data (e.g., `x_iter.buff`). This abstraction allows seamless integration with the concretized proofs from the algorithmic description, such as `big_add_correct` from Lst. 5.12, since both operate on the same high-level sequence representation.

5.3 Leveraging the Abstractions in Practice

To evaluate the effectiveness of Galápagos, we apply it to a real-world use case, OpenTitan [129]. OpenTitan is a TPM-like [157] chip that servers as a silicon root of trust. OpenTitan bootstraps its trust from a secure boot process [64, 133], which loads and executes properly signed firmware only. The signature scheme is RSA-3072, which is our first case study algorithm.

OpenTitan includes both a 32-bit RISC-V main core and a custom 256-bit big-number accelerator (dubbed the OTBN). For extra resiliency, OpenTitan supports secure boot with and without the accelerator enabled. Hence, we use Galápagos to produce verified implementations of RSA-3072 for both the RISC-V and OTBN. Our verified code has been burnt into the mask ROM currently in use for fabricating OpenTitan chips — the first instance, to our knowledge, of formally verified cryptographic code baked into hardware at scale.

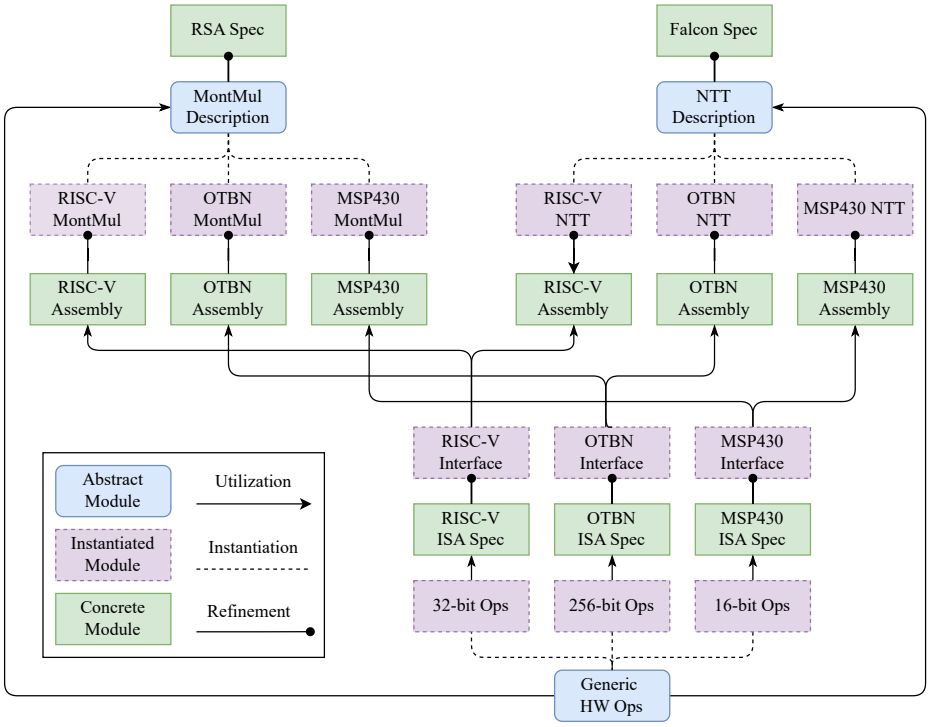


Figure 5.1: **Case Studies in Galápagos.**

To further validate Galápagos's ability to support heterogeneous hardware, we developed

an implementation for yet another architecture, MSP430 [21], in less than a week. MSP430 is a 16-bit micro-controller architecture designed by Texas Instruments for low-power embedded devices. We avoided ARM and x86, since they are also quite standard and well-studied in prior work [2, 3, 23, 25, 35, 138, 140].

To further validate Galápagos's ability to support diverse algorithms, we developed verified implementations of Falcon-512 [62], a post-quantum signature scheme standardized by NIST. To our knowledge, these are the first formally verified implementations of Falcon. Unlike RSA, Falcon is rooted in lattice-based cryptography. At the heart of our Falcon implementations is an NTT functor, constructed based on the algorithmic description in Alg. 2. Leveraging the functor support in Galápagos, we parameterized our NTT proofs over a polynomial ring, ensuring their reusability across other post-quantum algorithms.

5.3.1 Simplifying the Hardware Specification

Our case studies target OTBN, RISC-V, and MSP430, three ISAs with distinct bit-widths, addressing modes, and arithmetic operations. We define their formal executable semantics in Dafny. While these semantics are trusted, we bolster confidence in their correctness by running fuzz tests that compare their outputs against those of reference simulators.

The use of abstract operations (Sec. 5.2.3) in the instruction set architecture (ISA) specification is straightforward, so we focus on the usage of the memory interface. Overall, Galápagos' memory abstraction seamlessly supports all three architectures, despite differences in word sizes and addressing modes. We now discuss the architectures in more detail.

RISC-V.

RISC-V is an open standard ISA family [168]. For our case study, we use RV32IM, which is the 32-bit base integer ISA (47 instructions) with extensions for integer multiplication and division (8 instructions). The instruction set is quite standard, with a 32-bit address space and byte-addressable memory. There are only three data addressing modes: register, immediate, and indexed. One interesting wrinkle is that, unlike most platforms (including our other two), RISC-V does not have a dedicated flags register for zero, overflow, or sign bits; instead, the developer is expected to check for such conditions using standard ALU operations.

For the memory interface, we demonstrated how to wrap the `RV_LW` instruction using `read_iter` in Lst. 5.11, which leverages the `read_iter_preserves_inv` lemma provided by Galápagos. Although `RV_LW` supports only the register-plus-immediate addressing mode, it integrates smoothly with the iterator interface: either by combining `read_iter` with an explicit `addi` instruction to increment the pointer, or by setting the `inc` flag to `false` to avoid automatic pointer advancement.

MSP430.

MSP430 is a micro-controller family developed by Texas Instruments. It offers a minimalist 16-bit ISA with only 27 instructions (omitting, for example, multiplication). MSP430 memory is byte addressable, and its instructions have six possible addressing modes: register, indexed, absolute, indirect register, indirect auto-increment, and immediate.

The indirect auto-increment mode in the MSP430 uses a register operand as a pointer and increments the pointer after performing the load. This matches the programming pattern that moves the iterator of an array to the next entry after reading the current entry, making it a natural fit for Galápagos' iterator abstraction.

OTBN.

OTBN is a cryptographic accelerator ISA from the OpenTitan project. OTBN operates on 32 control registers, each 32 bits wide, and 32 data registers, each 256 bits wide. Hence, the data registers alone can potentially hold 1KB of data without any memory accesses. OTBN is designed to accelerate cryptographic computations involving large integers, such as those used in RSA or elliptic curve cryptography. OTBN supports 57 instructions, many of which offer configurable options. For example, the `BN.MULQACC` instruction performs a quarter-word (64-bit) multiplication and then adds the result to a dedicated accumulation register. The instruction can be customized to choose different quarter words from each source/destination register, to shift the multiplication result before accumulating it, and to clear the accumulation register before adding the result.

For the data-memory instructions, `BN.LID` and `BN.SID`, a control register provides the index of the data register as an operand, indirectly reading and writing the wide registers. The instructions read/write 256 bits of data memory and support indirect addressing modes with auto-increment. The full syntax of the load instruction is as follows:

```
BN.LID <grd>[<grd_inc>], <offset>(<grs>[<grs_inc>])
```

Both `grd` and `grs` are 32-bit control registers, where `grd` specifies the index of the wide register to use as a destination, and `grs` along with the `offset` specifies the source memory address. Suppose that `grd` is register `x1`, which contains the value `0x3`, and `grs` is register `x16`, which contains the value `0x8000`. With no offset, this instruction will load the 256-bit word at address `0x8000` into data register `w3`. Notably, the optional auto-increment feature aligns well with the `read_iter` iterator pattern, enabling efficient integration with Galápagos' abstractions.

	ISA Specification		High-Level Interface		Tally	
	Loc	Saved Ratio	Loc	Saved Ratio	Loc	Saved Ratio
Generic	453	-	1,140	-	1,593	-
MSP430	490	48%	1,091	51%	1,581	50%
RISC-V	685	39%	1,273	47%	1,958	44%
OTBN	1,506	23%	1,843	38%	3,349	32%

Table 5.2: **Simplifying the Hardware Specification with Galápagos.**

In Tab. 5.2, we summarize the lines of code developed for the ISA specifications and the high-level memory interface. The generic row contains the abstract operations and memory interface in Sec. 5.2.3, which is a one-time cost covered by Galápagos. The other rows show the additional lines of code needed to support each ISA's specification and abstraction. OTBN requires slightly more effort due to the complexities of the ISA. For the simpler ISAs, it saves up to half of the code that would have been written if developed without Galápagos.

5.3.2 Simplifying the Algorithmic Refinement

RSA-3072.

RSA signatures are simple to specify in terms of modular exponentiation of integer values. RSA implementations, however, are amenable to a wide variety of algorithmic and assembly-level optimizations. The algorithmic optimizations are quite complex to reason about even in isolation, let alone in the midst of a complicated assembly-level implementation. Hence Galápagos' split of these obligations between the algorithmic description and the hardware-specific implementation simplifies our correctness proofs.

Algorithmic Description. Following OpenTitan's unverified baselines, our algorithmic description employs the Montgomery multiplication (Alg. 1) for efficient modular exponentiation. Notably, Alg. 1 (and our algorithmic description) is parameterized over both by the radix (e.g., the machine-word's upper limit) and by the size of the big integers, which are represented by sequences of machine words, matching the multi-limb sequences in Sec. 5.2.4.

Another important detail is that Line 3 accumulates an intermediate result and requires several multi-limb operations (e.g., $u \cdot m$ is a product between a multi-limb sequence m and a machine word, which produces a multi-limb result, and similarly for $x[i] \cdot y$). Therefore, in the full algorithmic description, this line translates into a loop, performing the element-wise products and sums.

In terms of the algorithmic correctness, the main loop starting on Line 1 has the following two invariants:

$$a \equiv x[..i] \cdot y \cdot b^{-i} (\text{mod } m)$$
$$a < 2m$$

These invariants, along with the conditional subtraction at Line 6 of the algorithm, ensure overall correctness, i.e., the output is congruent to $x \cdot y \cdot b^{-n}$, and it is bounded by m. The

proof of the congruence invariant roughly follows these steps:

$$
\begin{aligned}
& (a + x[i] \cdot y + u \cdot m)/b && \pmod{m} && (5.1)\\
\equiv\ & (a + x[i] \cdot y + u \cdot m) \cdot b^{-1} && \pmod{m} && (5.2)\\
\equiv\ & (a + x[i] \cdot y) b^{-1} && \pmod{m} && (5.3)\\
\equiv\ & (x[..i] \cdot y \cdot b^{-i} + x[i] \cdot y) \cdot b^{-1} && \pmod{m} && (5.4)\\
\equiv\ & (y \cdot (x[..i] \cdot b^{-i} + x[i])) \cdot b^{-1} && \pmod{m} && (5.5)\\
\equiv\ & (y \cdot x[..i+1] \cdot b^{-i}) \cdot b^{-1} && \pmod{m} && (5.6)\\
\equiv\ & y \cdot x[..i+1] \cdot b^{-(i+1)} && \pmod{m} && (5.7)
\end{aligned}
$$

We note that the least significant word of $a + x[i] \cdot y + u \cdot m$ is 0, which justifies (5.2), and the evaluation rule of multi-limb numbers justifies (5.6). Moreover, the congruence relation fits perfectly into the subset handled by our `gbassert` extension to Dafny (Sec. 3.2.2).

Meanwhile, for the bound invariant, we instead rely on lemmas about non-linear arithmetic from our Dafny standard library (Sec. 5.2.1). The main steps are as follows:

$$
\begin{aligned}
& (a + x[i] \cdot y + u \cdot m)/b\\
\leq\ & (2m - 1 + x[i] \cdot y + u \cdot m)/b\\
\leq\ & (2m - 1 + x[i] \cdot (m-1) + u \cdot m)/b\\
\leq\ & (2m - 1 + (b-1)(m-1) + (b-1) \cdot m)/b\\
=\ & (2b \cdot m - b - 1)/b\\
<\ & 2m
\end{aligned}
$$

Algorithm 1 Montgomery Multiplication

Require:
b is some radix
n is some length
m, x, y are vectors length n with elements bounded by b
a is a vector length $n + 1$ with all 0 elements
$0 \leq x, y < m$
$m' = -m^{-1} \bmod b$

Ensure:
$a = x \cdot y \cdot b^{-n} \bmod m$

1: **for** $i = 0;\ i < n;\ i = i + 1$ **do**
2: $\quad u = (a[0] + x[i] \cdot y[0]) \cdot m' \bmod b$
3: $\quad a = (a + x[i] \cdot y + u \cdot m)/b$
4: **end for**
5: **if** $a > m$ **then**
6: $\quad a = a - m$
7: **end if**

Assembly Implementations. We ported the existing, unverified RSA-3072 implementations for RISC-V and OTBN into Vale. For the MSP430, we compiled a C version and transcribed the resulting assembly to Vale while making performance optimizations. For our proofs, we instantiate the algorithmic description's functor with hardware-specific modules that specify an appropriate radix for each platform (e.g., 2^{16} for the MSP430).

Given the lemmas instantiated from the algorithmic description, proving the correctness of the hardware-specific implementations was relatively straightforward, mostly boiling down to proving various hardware-specific bit-fiddling optimizations. The OTBN implementation was relatively easy, since it could fit all of the RSA integers entirely into registers. Its two sets of flag registers simplified carry propagation, and the built-in accumulator register likewise simplified the multi-word computations. The most significant proof challenge was proving that the implementation correctly used a chain of the (very complex) `BN.MULQACC` instruction to compute the product of two 256-bit numbers.

The MSP430 and RISC-V implementations resemble one another. Compared to OTBN, both support a simpler multiplication instruction, while RISC-V was complicated by the lack of a flags register.

Falcon-512.

To validate that Galápagos is applicable to other algorithms, we have used it to produce verified implementations of Falcon. Falcon is based on lattices and its security reduces to the short integer solution problem [1], which differs drastically from RSA.

The spec for Falcon is relatively concise, although still more verbose than RSA, since it depends on definitions of polynomial arithmetic. Simplifying a bit, Falcon verifies a signature s over (hashed) message m, using public key pk, by computing

$$s' \leftarrow m - s \cdot pk \bmod q$$

and checking that the distance between s and s' is small. The signature and the public key are encoded as polynomials, so the most computationally intense operation is computing the polynomial multiplication (i.e., $s \cdot pk$).

Algorithmic Description. Naively, a polynomial multiplication takes $O(N^2)$ time, but this can be optimized to $O(N \log N)$ using the number theoretic transform (NTT). In our abstract implementation, we employ the Cooley-Tukey (CT) butterfly algorithm [40] to compute a forward NTT operation (shown in pseudocode in Alg. 2). Notice that the algorithm, like our abstract implementation, is parameterized over the prime q that defines the field and the size n of the polynomials. Hence, our generic NTT implementation can be instantiated for many other lattice-based algorithms beyond Falcon.

While the pseudocode in Alg. 2 is relatively succinct, the justifications for why each step computes the right value are surprisingly subtle and are described across multiple research papers [102, 103, 119, 120]. We provide some intuitions for the algorithm's correctness here. The NTT algorithm works with a sequence of words, where each word represents a polynomial coefficient in the ring $\mathbb{Z}_q$. Hence we can think of a sequence as a polynomial

Algorithm 2 NTT with CT butterfly

Require:
n is a power of two.
q is a prime such that $q \equiv 1 (\text{mod } 2n)$.
a is a vector in $\mathbb{Z}_q^n$ (standard order).
ψ is a primitive $2n$-th root of unity in $\mathbb{Z}_q$
Ψ_{rev} is a vector in $\mathbb{Z}_q^n$ with powers of ψ (bit-reversed order).
Ensure:
a is the NTT of its initial content (bit-reversed order).

1: $t = n$
2: **for** $m \leftarrow 1;\ m < n;\ m \leftarrow 2 \cdot m$ **do**
3: $\quad t = t/2$
4: $\quad$ **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
5: $\quad\quad s = \Psi_{rev}[m + i]$
6: $\quad\quad$ **for** $j \leftarrow 2i \cdot t;\ j < 2i \cdot t + t\ ;\ j \leftarrow j + 1$ **do**
7: $\quad\quad\quad e = a[j]$
8: $\quad\quad\quad o = a[j + t] \cdot S$
9: $\quad\quad\quad a[j] = (e + o) \mod q$
10: $\quad\quad\quad a[j + t] = (e - o) \mod q$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end for**

and reason about the effect of evaluating it on a point. If we have sequence $a \in \mathbb{Z}_q^n$ and point $x \in \mathbb{Z}_q$, then the evaluation $a(x)$ can be written as:

$$\sum_{j=0}^{n-1} a[j] \cdot x^j$$

Let ω be the primitive n-th root of unity in the ring $\mathbb{Z}_q$. The NTT algorithm evaluates the polynomial a at the points $\omega^0, \omega^1..\omega^{n-1}$. More formally:

$$\text{NTT}(a)[i] = \sum_{j=0}^{n-1} a[j]\omega^{ij}$$

The CT butterfly optimization uses the fact that polynomial evaluation can be split into the evaluation of the terms corresponding to even and odd powers. Let the corresponding coefficients be a_e and a_o, then we can rewrite $a(x)$ as $a_e(x^2) + x \cdot a_o(x^2)$. This reduces the problem to to evaluating the polynomials a_e and a_o on the points $\omega^0, \omega^2..\omega^{2(n-1)}$. Since ω is a primitive n-th root, the list now only contains $\frac{n}{2}$ distinct points. Applying this recursively produces the $O(N \log N)$ running time.

For additional efficiency, Alg. 2 is an iterative and in-place version of the CT butterfly. The loop over m that starts on Line 2 corresponds to the size of the polynomial, which doubles at each level. The loops over i and j combine the evaluations of the smaller polynomials.

Assembly Implementations. Having dealt with the complex mathematical reasoning in our abstract implementation, our concrete Falcon implementations focus on proving that they faithfully execute the operations dictated by the abstract implementation. Of the three implementations, the OTBN one is the simplest, since we were able to implement Falcon's many additions and subtractions modulo q by simply loading q into OTBN's dedicated modulus register and then invoking OTBN's modular addition and subtraction instructions.

```
add     a1, a1, a0 ; sum up a0 and a1
sltu    a0, a1, a0 ; if the sum is less than a0, set a0
```

Listing 5.14: **RISC-V Extract Carry Bit**

Implementing these operations on the MSP430 and RISC-V involves some non-trivial bit manipulation. For example, on RISC-V the carry bit can be extracted through conditional branches, but Lst. 5.14 is more efficient.

```
CLR R10          ; clear R10
SUBC R10, R10    ; subtract with overflow flag
                 ; R10 is either 0x0000 or 0xFFFF
AND 12289, R10 ; R10 is conditionally set to Q
```

Listing 5.15: **MSP430 Set on Overflow**

Lst. 5.15 shows an example for MSP430. Without using branches, the code conditionally sets `R10` to `12289` (the modulus q) based on the overflow flag.

Tab. 5.3 presents the lines of code developed for our RSA and Falcon implementations. The specification and the generic implementation are the per-algorithm one-time cost. We note that the generic implementation for RSA is much shorter than Falcon's, largely due to the Dafny standard library's support for big-integer reasoning. For the concrete implementations, the Vale code embeds the concrete assembly while the Dafny code measures the additional platform-specific lemmas needed. The generic code reduces the proof burden for RSA by ~30% for and for Falcon by more than 60% (RSA has a lower ratio due to its heavy use of our standard library). In our initial verification efforts, we verified implementations of RSA for the OTBN and RISC-V using traditional monolithic techniques from prior work [11, 24, 55]. Motivated by the duplication across these implementations, we then developed the Galápagos framework and used it to refactor the code. This reduced the developer-written platform-specific code by 28% for OTBN and 29% for RISC-V. We further leveraged the framework to both specify the MSP430 and add a custom RSA implementation, in approximately one week of developer effort.

5.3.3 Meeting the High Performance Requirements

We evaluate the performance of our verified RISC-V and MSP430 implementations on physical development boards, comparing their cycle counts against unverified baselines. For RISC-V, we use SiFive's HiFive1 Rev B, featuring the Freedom E310 microcontroller,

	RSA			Falcon		
	Dafny	Vale	savings	Dafny	Vale	savings
Spec	58	-	-	440	-	-
Generic	963	-	-	5,280	-	-
MSP430	32	1,757	34%	290	2,945	62%
RISC-V	446	1,824	29%	543	2,654	62%
OTBN	339	2,103	28%	163	2,641	65%

Table 5.3: **Simplifying the Algorithmic Refinement with Galápagos.**

running at its default clock speed of 16 MHz. For MSP430, we utilize a Texas Instruments LaunchPad equipped with the MSP430FR2476 microcontroller, configured to operate at 8 MHz.

As OpenTitan chips were still undergoing their initial production run during this work, we rely on OpenTitan's cycle-accurate simulator [128] to evaluate the performance of our OTBN implementation.

Unverified Baselines. For RSA, prior to our work, the OpenTitan team developed a hand-written assembly implementation for OTBN and used a C compiler (configured to optimize for size) to generate code for RISC-V. Similarly, we used a C compiler to produce the MSP430 implementation. These three implementations serve as the unverified RSA baselines. For Falcon, while a pre-existing C implementation [86] exists, there are no optimized assembly implementations for the hardware platforms we target. Consequently, we rely on a C compiler to generate unverified baselines for RISC-V and MSP430. Since no unverified baseline exists for OTBN, we developed our verified implementation from scratch.

	MSP430	**RISC-V**	**OTBN**
RSA			
Baseline	144,998,445	9,355,922	160,814
Verified	142,870,737	9,454,635	160,664
% Change	-1.47%	+1.05%	0%
Falcon			
Baseline	2,810,513	846,946	-
Verified	2,015,556	846,926	256,796
% Change	-28.3%	0%	-

Table 5.4: **Performance of Verified Implementations.**

Tab. 5.4 shows the cycle counts for our verified implementations and their unverified baselines. We find that our verified implementations typically perform within ±2% of their respective baseline. Our verified Falcon implementation for the MSP430, however, is considerably faster than its compiled baseline. We believe this is due to the fact that we

came up with a much more efficient register allocation than the compiler.

5.4 Work Status and Personal Contribution

Galápagos was published at CCS'23 [174]. I was the lead author of the paper, where I did most of the paper writing, experimented with multiple iterations of the Galápagos framework, and implemented most of the case studies. In particular, as simple as Alg. 2 might look, the NTT functor was one of the hardest thing I had to prove correct. I ended up using a Python script to empirically "guess-and-check" the invariants for the nested loops, one level at a time. I am still not sure if I completely understand the bit-reversal ordering, but I was able to prove the algorithmic description correct using Dafny.

My advisor Bryan Parno worked on the implementation of the verified functors, which I did not contribute to. Sydney Gibson contributed to some of the case studies for RSA. Sarah Cai and Menucha Winchell were the main contributors to the Dafny standard library, when Sydney Gibson and I were their mentors for their summer internship at CMU.

Chapter 6

Stabilizing Proofs

Proof instability is a phenomenon where trivial changes to source code may lead to spurious verification failures. In our earlier discussion (Sec. 2.3.2), we highlighted how instability presents a unique challenge to APV, significantly limiting the scalability of verification. In this chapter, we discuss our line of work to measure, understand, mitigate, and repair instability in APV.

While the program verification community has recognized the issue of instability [52, 79, 96], there is no methodical way to detect it, let alone to mitigate or repair it. In the SMT community, SMT-COMP [13], the annual competition for SMT solvers, does not include categories to deter instability. Possibly as a result, the stability of some APV projects actually deteriorates with solver upgrades (Sec. 6.1.4).

We therefore start with a systematic study of the instability phenomenon in Sec. 6.1. informing both the APV and SMT communities with empirical data and statistical analysis. In Sec. 6.1, we introduce Mariposa, a tool to detect instability in APV queries. We detail our methodology, including our mutation-based tests in Sec. 6.1.1, metrics of instability in Sec. 6.1.2, and taxonomy of stability status in Sec. 6.1.3. We then shed light on the current ecosystem with experimentation over a substantial number of APV projects in Sec. 6.1.4.

In Sec. 6.2, we present our work on instability mitigation with Shake, a context-pruning technique we apply at SMT preprocessing. In Sec. 6.2.1, we leverage Mariposa for controlled experiments, and we find that irrelevant query context is a major cause of instability. In Sec. 6.2.2, we present a theorem-proving view on APV queries, decomposing a query into a verification goal and a set of supporting axioms. In Sec. 6.2.3, we leverage this theorem-proving perspective in the design of Shake, which reduces the number of irrelevant axioms in the context to improve stability.

In Sec. 6.3, we present Cazamariposas, a tool to repair instability. Cazamariposas goes a step further than Shake, pinpointing the specific axioms causing instability, while also providing a repair strategy to stabilize the query. In Sec. 6.3.1, we discuss our methodology to test the stability impact of each axiom individually, leveraging the solver-produced proof

logs. In Sec. 6.3.2, Sec. 6.3.3, and Sec. 6.3.4, we discuss our methodology to effectively triage the axioms using novel proof and trace mining techniques, so that we put the most likely suspects under investigation, and we can identify the problematic axioms with minimal overhead.

6.1 Measuring Instability with Mariposa

In this section, we outline our methodology to detect and measure proof instability in Mariposa, which we then leverage to understand the state of the ecosystem.

For a given query-solver pair (Φ, s), Mariposa answers two conceptual questions: (1) Is Φ stable under s? and (2) How stable or unstable is it? Intuitively, instability means that the performance of s diverges when Φ undergoes seemingly-irrelevant mutations. Following the intuition, we describe the mutations in Mariposa and the rationales behind them in Sec. 6.1.1. We introduce metrics to quantify the stability/instability in Sec. 6.1.2. We then categorize the stability status of (Φ, s) in Sec. 6.1.3 based on the metrics.

6.1.1 Mutating the Input Query

In Mariposa, we focus on query mutations that not only preserve the *semantic meaning*, but also maintain *syntactic structure*. Here we give the (informal) definitions with respect to an *original* query Φ and its *mutant* Φ'.

- **Semantic Equivalence**. Φ and Φ' are semantically equivalent when there is a bijection between the set of proofs for Φ and the set of proofs for Φ'. In other words, a proof of Φ can be transformed into a proof of Φ', and vice versa.
- **Syntactic Isomorphism**. Φ and Φ' are syntactically isomorphic when there is a bijection between the symbols (e.g, functions and sorts), as well as the commands (e.g., assertions). In other words, each symbol or command in Φ has a counterpart in Φ', and vice versa.

Given the definitions, it should be reasonable to expect that Φ and Φ' have similar performance on the same solver s. For our experiments, we are interested in mutation that also corresponds to common developer practices. Specifically, we consider the following three methods:

- **Assertion Shuffling**. It is common to reorder source-level procedures, which roughly corresponds to shuffling the order of commands in the generated SMT query.
- **Symbol Renaming**. It a common practice to rename source-level procedures, types, or variables, which roughly corresponds to SMT-level α-renaming.
- **RNG Reseeding**. SMT solvers optionally take as input a random seed, which influences some of the internal non-deterministic choices. The seed has no effect on

the query's semantics but is known to affect the solver's performance[1].

Conceptually, when we apply a mutation method to a query Φ in an exhaustive manner, we obtain a set of mutants M_Φ. Consider assertion shuffling as an example. If Φ contains 100 assertions, then M_Φ would have $100! \cong 9 \times 10^{157}$ permutations of Φ, including Φ itself.

6.1.2 Quantifying the (In)Stability

Intuitively, whether a query-solver pair (Φ, s) is stable depends on how s performs on M_Φ. We thus introduce two metrics to quantify the degree of instability/stability of (Φ, s) based on the performance of s on M_Φ.

Metric: Mutant Success Rate. We denote the metric with $r_{\Phi,s}$, which is the ratio of the number of mutants that s can prove in M_Φ. That is, we do not take into account the type of failure (i.e., `unknown` and `timeout`), or how long it takes to prove a mutant. Intuitively, the ratio reflects the consistency of verification results (and only results) from s. Therefore, a low value of $r_{\Phi,s}$ indicates consistent verification failures; a high value of $r_{\Phi,s}$ indicates consistent verification successes; and a value of $r_{\Phi,s}$ in between indicates instability.

Metric: Mutant Time Deviation. We denote the metric as σ_Φ, which is the standard deviation of the response time of s over M_Φ. The metric reflects the variation in the verification time (regardless of the result) from s. Intuitively, a large value of $\sigma_{\Phi,s}$ means the developer cannot expect a consistent response time from s, which is generally undesirable.

We note that consistency here does not mean the same as consistency in a logical system. Due to the undecidable nature of the underlying logic, it may be perfectly reasonable for a solver to return different results (e.g., `unsat`, `unknown`, or `timeout`) on different members of M_Φ. However, returning `unsat` and `sat` on different members of M_Φ would be logically inconsistent. Fortunately, we never encounter such cases in our experiments.

We mainly use $r_{\Phi,s}$ to characterize the stability of (Φ, s), and we use $\sigma_{\Phi,s}$ to complement the analysis *only* when $r_{\Phi,s}$ is high. In this way, we prioritize the consistency of the verification outputs over that of the response times. Meanwhile, when $r_{\Phi,s}$ is high (i.e., the verification outputs are consistent), the larger σ_Φ is, the less stable (Φ, s) actually is.

[1] Historically, some verification tools have attempted to use reseeding to measure instability. For example, F* have options to run the same query multiple times with different random seeds and report the number of failures encountered. Dafny has recently started to perform shuffling and renaming, where its option has changed from `randomSeedIterations` to `randomizeVcIterations`.

6.1.3 Determining the Stability Status

We further introduce four stability categories, following the intuition behind the success rate $r_{\Phi,s}$. To simplify the discussion, we first assume that M_Φ is generated by a single mutation method, and then discuss how to combine results from multiple mutations. We include two parameters to the scheme: r_{sol} and r_{stb}, which correspond respectively to the lower and upper bounds of the success rate range for unstable queries.

Category	Interval	Description
`unsolvable`	$[0, r_{sol})$	s fails to prove most mutants of Φ
`unstable`	$[r_{sol}, r_{stb}]$	s fails to consistently prove mutants of Φ
`stable`	$(r_{stb}, 1]$	s consistently prove mutants of Φ
`inconclusive`	N/A	no statistical significance

Table 6.1: **Mariposa Stability Categories**

6.1.3.1 Sampling the Mutants

In practice, it is often intractable to enumerate all members of M_Φ (recall the 100! mutants from our shuffling example), so the true value of $r_{\Phi,s}$ is generally unknown. Therefore we resort to statistical tests to estimate $r_{\Phi,s}$ from a sample set of mutants $\hat{M}_\Phi \subseteq M_\Phi$. We use $r_{\hat{\Phi},s}$ to denote the observed sample success rate.

Since we are estimating population proportions, the Z-test [58] is a natural choice. The Z-test is a statistical test that determines whether the proportion of a sample is significantly different from a hypothesized proportion. The test is parameterized by the α value, which specifies confidence in the judgment. We use an $\alpha = 0.05$ (i.e., 95% confidence), which is a standard choice.

Fig. 6.1 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., $r_{\Phi,s} < r_{stb}$. 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.

6.1.3.2 Accounting for Timeouts

Since there is no guarantee that a solver will terminate, we impose a time limit T_{lim} on all of our experiments. We note that solvers can also bound the execution with a resource limit (`rlimit`) instead of a time limit, in an effort to make results more consistent across

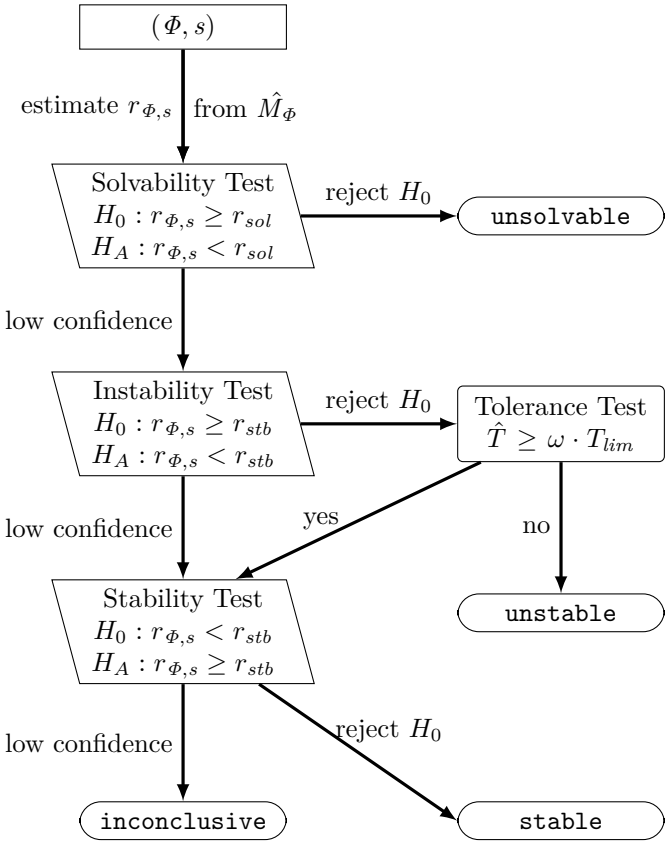


Figure 6.1: **Mariposa Categorization Flowchart**

different computing platforms. However, the resource tracking often counts only some of the resources used (e.g., it may ignore the resources spent inside a theory solver). Further, there is no guarantee of consistency across solver versions, let alone across different solvers. Hence, Mariposa uses execution time as a more universal measure.

Mariposa considers a mutant that times out a verification failure. However, when the expected response time of M_Φ 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 scheme with a tolerance factor ω between 0 and 1. When Mariposa observes mixed results in $\hat{M}_\Phi$, it estimates the expected response time for M_Φ, using the mean response time of successful samples, denoted as $\hat{T}$. If the latter is close to the time limit, i.e., $\hat{T} \geq \omega \cdot T_{lim}$, the failures may be due to an insufficient T_{lim}. In that case, we take a conservative approach and do not label (Φ, s) as `unstable`. Fig. 6.1 shows the tolerance test in the workflow.

6.1.3.3 Accounting for Different Mutations

We have based our discussion so far on a single mutation method. In our experiment, we apply **shuffling**, **renaming**, and **reseeding**, each outputs a stability category. 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`.

We 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 Sec. 6.1.4 we show how mutation methods differ in their ability to detect instability.

6.1.4 Characterizing the Ecosystem

We materialize our methodology in the Mariposa tool, which we then leverage in large-scale empirical studies to characterize the state of (in)stability. Here we first introduce the query benchmarks we curated and the general experiment setup. We then discuss the results that directly reflect upon the ecosystem, which also serves as the baseline for our later work on mitigation and repair. We defer further controlled experiments using Mariposa to Sec. 6.2 and Sec. 6.3.

6.1.4.1 Query Benchmarks

We select existing system verification projects in the literature as our benchmarks. We generally follow these guidelines with respect to the selection: (1) The source code of the project is publicly available, so that the queries and experiments can be reproduced. (2) The project required substantial effort, so that the queries are representative of real-world use cases. (3) The work (or its non-verified baseline) has been published at a peer-reviewed venue, so that the results are likely to be of interest to the community.

We have curated two benchmark suites: the Mariposa Bench and the Verus Bench. We introduced the Mariposa Bench in 2023, with a focus on system verification efforts using F* or Dafny, comprising six projects.

- DICE$^\star_F$ [155] is an implementation of the DICE boot protocol [105] using F*.
- Komodo$_D$ [59] is a security hypervisor written in Dafny.
- Komodo$_S$ [122] is a re-implementation of Komodo$_D$ using Serval.
- VeriBetrKV$_D$ [72] is a key-value store written in Dafny.
- VeriBetrKV$_L$ [100] is a re-implementation of VeriBetrKV$_D$.
- vWasm$_F$ [26] is an implementation of a WebAssembly [70] compiler using F*.

While Mariposa Bench is by no means an exhaustive collection of APV queries, it is the first benchmark of the kind. We also decided to include legacy projects even if they are no longer maintained, (e.g., Komodo$_D$ is ~8 years old), so that we can study the longitudinal evolution of the ecosystem.

After our initial stability study in 2023, the Verus language ecosystem gradually matured. We subsequently curated the Verus Bench in 2025, where we included more recently-developed Verus-based projects.

- Atmosphere [34] is a full-featured microkernel.
- Anvil [152] is a tool for verifying Kubernetes controllers. The work includes three verified controllers: ZooKeeper, RabbitMQ, and FluentBit.
- IornKV$_V$ [159] is a re-implementation of the IronFleet [78] distributed key-value store.
- Splinter$_V$ [163] is an implementation of the SplinterDB [38] key-value store.
- VerusMimalloc [160] is a concurrent memory allocator.
- VerusNR [161] is a concurrent NUMA-aware node-replication library [32].
- VerusStorage [164] is a storage system targeting persistent memory devices.
- VerusPT [162] is an implementation of the page table in NrOS [19].
- Verismo [176] is a security module for confidential VMs.

We summarize the source-program line counts and SMT query counts in Tab. 6.2. For the ease of analysis, we have slightly modified the project names, distinguishing alternative implementations of the same underlying system.

It is worth noting that we used slightly different configurations for the two benchmarks in the experiments. Specifically, we set a 60-second timeout for Mariposa Bench, but we use a 10-second timeout for Verus Bench. Our choices are consistent with the build

Project	Source	Queries
DICE$^{\star}_{F}$	~24.7 kLoc	1,536
Komodo$_{D}$	~25.8 kLoc	2,054
Komodo$_{S}$	~3.7 kLoc	773
VeriBetrKV$_{D}$	~44.7 kLoc	5,325
VeriBetrKV$_{L}$	~49.0 kLoc	5,600
vWasm$_{F}$	~14.8 kLoc	1,755
Total	-	17,043

Project	Source	Queries
Atmosphere	~18.4 kLoc	331
Anvil	~30.2 kLoc	1,808
IronKV$_{V}$	~7.6 kLoc	363
Splinter$_{V}$	~25.8 kLoc	1,215
VerusMimalloc	~17.1 kLoc	725
VerusNR	~6.3 kLoc	254
VerusStorage	~4.9 kLoc	396
VerusPT	~6.8 kLoc	338
Verismo	~22.5 kLoc	2,126
Total	-	7,256

Table 6.2: **Projects in Mariposa/Verus Bench**

configuration of the projects. Verus-based projects in general are expected to have fast verification turnaround time, which is one of the design goals of the Verus language. We now discuss the experiment setup in more detail.

6.1.4.2 Experiment Setup

Machine Specification. We ran all the experiments on the same machine cluster, each with an Intel Core i9-9900K (max 5.00 GHz) CPU, 128 GB of RAM, and the Ubuntu 20.04.3 LTS operating system. While there are 16 hyper-threads available on each machine, we limited the number of threads to 7 to avoid overloading the CPU or causing excessive cache misses.

Mariposa Configuration. Configuring Mariposa is slightly non-trivial. The parameters offers a trade-off between computational resources and statistical confidence. Collectively, the sample size $\|\hat{M}_{\Phi}\|$, the thresholds (r_{sol}, r_{stb}), and observed success rate $r_{\hat{\Phi},s}$ determine the significance (i.e., the p-value) of a hypothesis test. Specifically, we can claim a statistically significant result only if $p < \alpha$, To simplify the discussion, we set $\alpha = 0.05$, which is fairly standard in the literature.

We have to balance the following considerations:

- We need to set reasonable thresholds so the result meaningfully reflects stability status. For example, $r_{stb} = 60\%$ is an overly generous threshold for `stable` queries. An observed $r_{\hat{\Phi},s} = 70\%$ may exceed r_{stb} in a statistical sense, but most developers would disagree that Φ is stable.
- We need to sample sufficiently large $\hat{M}_{\Phi}$ to avoid excessive number of inconclusive results. For example, suppose we set $r_{stb} = 95\%$ and $\|\hat{M}_{\Phi}\| = 3$. Even if we observe $r_{\hat{\Phi},s} = 100\%$, we cannot conclude `stable`, because $p \approx 0.34555$ is much larger than α.
- We also need to budget $\|\hat{M}_{\Phi}\|$ and T_{lim} so that the experiments finish in a reasonable

amount of time. Larger $\|\hat{M}_\Phi\|$ means more mutants to run per original query. Larger T_{lim} means potentially more time spent per mutant. Given that our original query count (24,599) is non-trivial, we need to be careful about these two multipliers.

We resolve the configuration problem with a sanity-check scenario for the Solvability Test. Specifically, when the observed success rate $r_{\hat{\Phi},s} = 0\%$ (i.e., all sample mutants failed), we should have enough confidence to conclude `unsolvable`.

Suppose we set $r_{sol} = 1\%$, we need at least $\|\hat{M}_\Phi\| = 268$ to draw such a conclusion ($p \approx 0.04995$). Assuming no machine crashes or disruptions (which happens more often than one might expect), this roughly translates to a few weeks (wall-clock) turnaround time for a particular solver s over the whole Mariposa Bench using our infrastructure. This is less than ideal, since we also had plans to experiment with multiple solvers.

On the other hand, if we were to relax $r_{sol} = 5\%$, $\|\hat{M}_\Phi\| = 60$ is more than enough to conclude ($p \approx 0.03778$). This reduces the turnaround time to a week or so, which is acceptable for our purpose. Due to the symmetry, if we set $r_{stb} = 95\%$, $\|\hat{M}_\Phi\| = 60$ is more than enough to conclude `stable` when the observed success rate is 100% (i.e., all sample mutants passed).

Parameter	Value	Description
α	0.05	Significance level for statistical tests
r_{sol}	0.05	Minimum success rate threshold
r_{stb}	0.95	Minimum stability rate threshold
$\|\hat{M}_\Phi\|$	60	Sample size for each mutation method
T_{lim}	60 (seconds)	Time limit for each mutant
ω	0.8	Tolerance factor for performance variability

Table 6.3: **Default Mariposa Configuration**

We summarize our default configuration of the Mariposa tool in Tab. 6.3, with the threshold settings $r_{sol} = 5\%$ and $r_{stb} = 95\%$, and a sample size of $\|\hat{M}_\Phi\| = 60$ for each mutation method (since we have 3 mutation methods, the total number of mutants per original query is 180). As we mentioned previously, for the Verus Bench, we adopt a more aggressive time limit of $T_{lim} = 10$ seconds.

Solver Selection. For most of our experiments, we focus on the Z3 SMT solver [47], which all of our experiment projects were developed with, except for Komodo$_S$, which used both Z3 and CVC4 [12]. While we provide some results on cvc5 in Sec. 6.2, we had initially planned to experiment with cvc5 more extensively.

Unfortunately, our preliminary experiments showed that the current APV ecosystem is over-fitted to Z3. An out-of-the-box cvc5 solver cannot parse any queries emitted from Dafny or F*. Due various bits of Z3-specific syntax and features these tools rely on, the resulting queries are not strictly SMT-LIB compliant.

After we converted the queries into standard SMT-LIB format, cvc5 could only solve

~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.

6.1.4.3 Recent Sanpshots

We first report the stability of both benchmarks on a recent version of Z3. As we show in Tab. 6.4, the amount of instability is generally small, but non-trivial nonetheless.

Project	**Unstable Queries** Z3 4.12.5
DICE$^\star_F$	20 (1.30%)
Komodo$_D$	93 (4.53%)
Komodo$_S$	4 (0.52%)
VeriBetrKV$_D$	172 (3.23%)
VeriBetrKV$_L$	256 (4.57%)
vWasm$_F$	4 (0.23%)

Project	**Unstable Queries** Z3 4.12.5
Atmosphere	1 (0.30%)
Anvil	20 (1.11%)
IronKV$_V$	0 (0.00%)
Splinter$_V$	2 (0.16%)
VerusMimalloc	7 (0.97%)
VerusNR	2 (0.79%)
VerusStorage	22 (5.56%)
VerusPT	2 (0.59%)
Verismo	13 (0.61%)

Table 6.4: **Recent Snapshot of Instability Status**

6.1.4.4 Longitudinal Results.

We are also interested in the historical status of stability in the APV ecosystem. Since the Verus Bench is relatively new, we perform this part of the study on the Mariposa Bench. In our Mariposa work (2023), we tested eight (now) legacy versions of Z3, with the earliest dating back to 2015. In particular, for each project we have included its *artifact solver*, which is the version used in the project's official artifact.

We organize our experimental results around a series of research questions (RQs). We present the results from a subset of projects here and defer the rest to the appendix.

RQ1. Do Solver Upgrades Improve Stability?

In Fig. 6.2, each stacked bar shows the proportions of categories in a project-solver pair. 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 artifact solver for each project is marked with a star ($\star$). In all project-solver pairs, the majority of queries are stable. However, a non-trivial amount of instability persists as well.

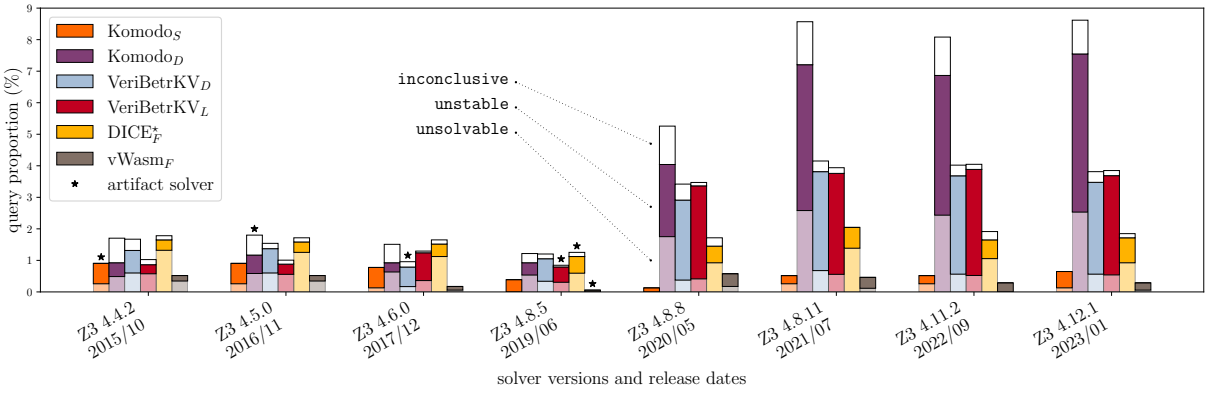


Figure 6.2: **Longitudinal Evolution of Stability Status**

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`.

Tab. 6.5 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.

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 6.5: **Regression Bisecting Z3 Commits**

Coincidentally, when we contacted the Z3 developers, 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 Z3's main branch.

RQ2. Do Projects Differ in Stability?

Komodo$_D$ vs. Komodo$_S$. The original Komodo$_D$ is a security hypervisor written in Dafny, which often generates undecidable queries. Komodo$_S$ is a re-implementation of Komodo$_D$ using Serval, which requires developers to work within a decidable of FOL. 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.

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 Z3 4.12.1. 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$_L$ vs. VeriBetrKV$_D$. As we discussed in Sec. 3.1.3, the two systems adopts different approaches to heap reasoning, where verification performance improves with linear types. However, the does not appear to generalize to stability: VeriBetrKV$_L$ is only slightly more stable than VeriBetrKV$_D$ when using their artifact solvers, and both suffer similar stability regressions on later solvers.

vWasm$_F$ vs. the rest. We notice that vWasm$_F$ is remarkably stable: the `unstable` proportion is almost negligible across all solver versions. Unlike Serval, F* is not limited to decidable logic, which makes the stability of vWasm$_F$ somewhat surprising. Since two authors of vWasm$_F$, Jay Bosamiya and Brayn Parno also worked on Mariposa, we get to document some of their manual engineering effort into stabilizing the the queries including the following 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) quantified formulas, and manually pick good trigger patterns. Particularly complex proofs necessitated even more drastic measures: using F* 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 tuning, these results suggest that the developers and/or the APV frameworks can potentially shape their queries to minimize or

control instability.

RQ3. Do Longer Time Limits Mitigate Instability?

As we discussed in Sec. 6.1.3, the choice of time limit T_{lim} 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 as of the time) with a limit of 150s (2.5 × 60s).

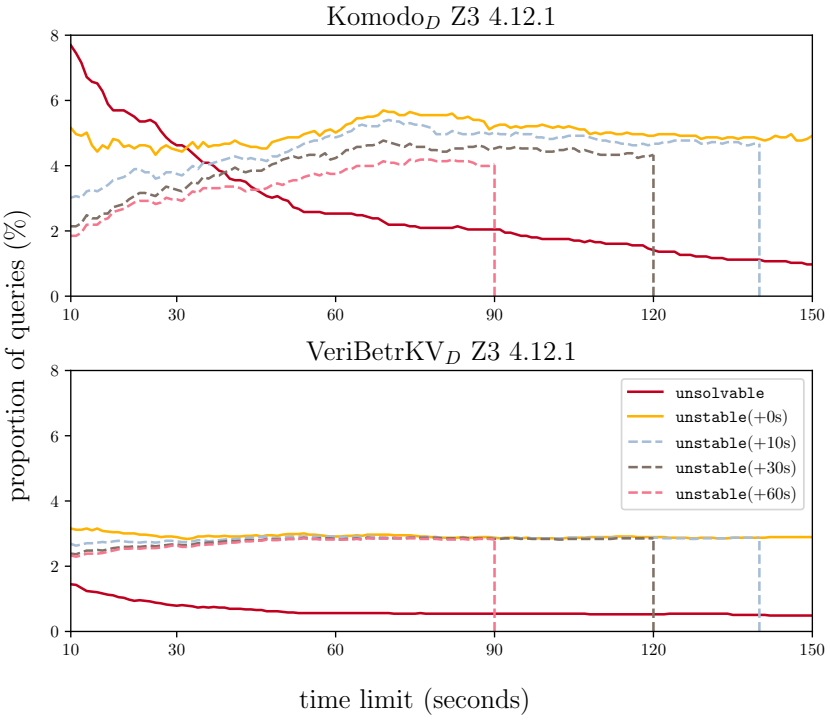


Figure 6.3: **Comparison on Time Limit Settings**

In Fig. 6.3 we report the proportion of `unsolvable` and `unstable` queries for each T_{lim} in Komodo$_D$ and VeriBetrKV$_D$. We observe that the `unsolvable` proportion drops as T_{lim} 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_{lim} and T_{lim} + step, for steps of 10, 30 and 60 seconds. One can interpret a T_{lim} + step curve as follows: if some queries are `unstable` at T_{lim}, it reports how many of them will remain `unstable` at T_{lim} + step.

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.

In Fig. 6.4, 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 Sec. 6.1.3. Since the choice of T_{lim} may also impact the categorization, we present results for different T_{lim} as well.

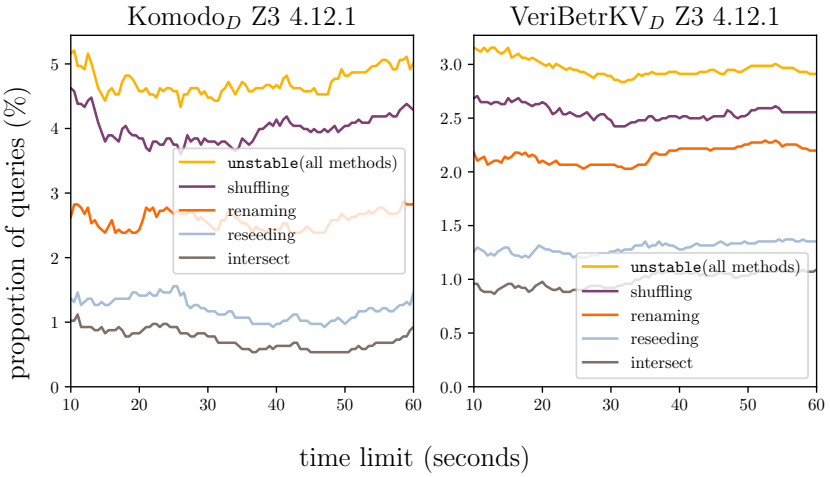


Figure 6.4: **Comparison on Mutation Methods**.

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_{lim}. 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?

In Sec. 6.1.2, we introduced the metric **Mutant Time Deviation**, where a large $\sigma_{\hat{\Phi},s}$ indicates less actual stability, even if mutants consistently succeed. Fig. 6.5 shows the distribution of $\sigma_{\hat{\Phi},s}$ in the `stable` queries, which are mostly less than 1s, but there are exceptions exceeding 10s, which is significant given the 60s limit. Mutation methods also differ in their impact.

RQ6. Is the Original Query Special?

In our methodology, we treat the original query 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 Fig. 6.6, we compare the verification time of the original query against the median of

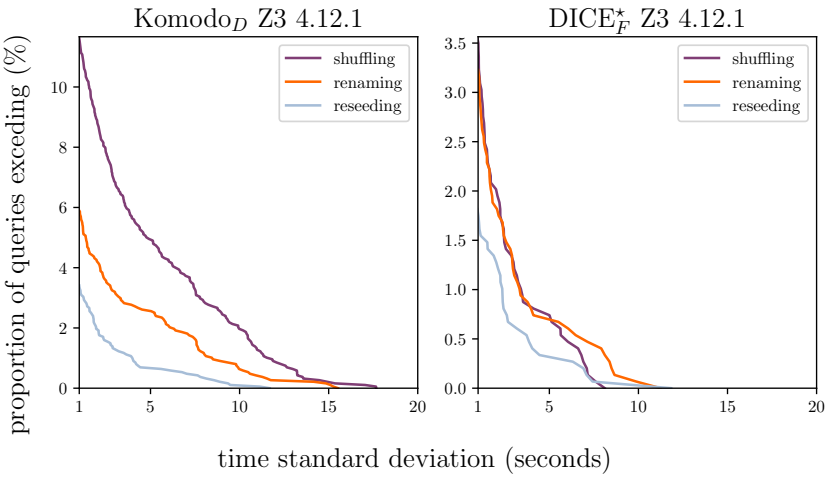


Figure 6.5: **Degree of Stability in Stable Queries**

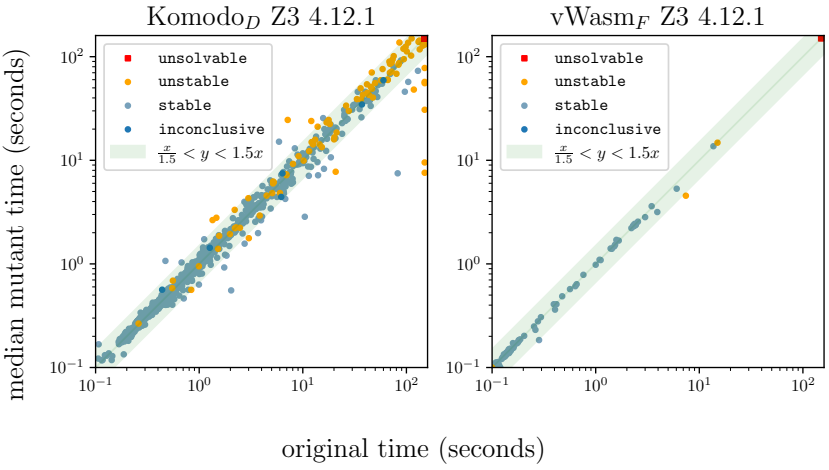


Figure 6.6: **Comparison on Original and Mutant Queries**

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 $\pm 50\%$ of each other. In vWasm$_F$, where the `unstable` proportion is the lowest, the two have nearly identical performance.

6.2 Mitigating Instability with Shake

Mariposa provides a systematic way to measure instability, which brings us a step closer to addressing the challenge. In this section, we investigate the irrelevant query context as a cause of instability, and discuss context pruning as a mitigation strategy.

In Sec. 6.2.1, we further leverage our Mariposa tool and benchmarks, conducting a large-scale controlled experiment on the unsatisfiable core. We find that typically $\geq 96.23\%$ of the query context is irrelevant, while accounting for $\sim$78.3% of the unstable instances.

Motivated by the findings, in Sec. 6.2.2, we propose a novel SMT context pruning technique, named Shake, to improve stability. We base Shake on the insight that APV queries are typically automated theorem proving (ATP) tasks [60], where each query is composed of a goal assertion along with axiom assertions. Shake triages the relevance of the axioms with respect to the goal, and prunes the less relevant axioms.

6.2.1 Characterizing the Query Context

In this section, we study the connection between query context and stability. For each query, we analyze its *unsatisfiable core*. As we discussed in Sec. 2.4, a given query generally does not have a unique unsatisfiable core. Te solver can produce one core unsatisfiable upon reaching an `unsat` result, Nevertheless, it is a useful approximation of the relevant assertions.

Briefly reviewing our notations: for a query $\Phi = \bigwedge_{i=0}^{n} \psi_i$, we use $\Gamma_\Phi = \{\psi_0, ..., \psi_n\}$ to denote the set of assertions in the context, and we use Φ_C to denote some unsatisfiable core of Φ, where $\Gamma_{\Phi_C} \subseteq \Gamma_\Phi$.

6.2.1.1 Exporting the Unsatisfiable Core

In theory, when we run a query Φ on an SMT solver s, we can simply export Φ_C with the `produce-unsat-cores` option enabled. In practice, obtaining Φ_C can sometimes be non-trivial, especially on unstable queries. Though uncommon, two types of problems may occur, and we document our workarounds here.

Unsuccessful Export. The solver s might not be able to produce a core for a given Φ, which can be due to several reasons.

- (Φ, s) is unstable. Specifically, s fails on Φ but succeeds on certain mutants of Φ.
- (Φ, s) behaves differently with core production. Specifically, s may report `unsat` on Φ, but returns `unknown` as soon as we enable core production.
- (Φ, s) is completely unsolvable (regardless of mutations). However, (Φ, s') might be solvable, where s' is a different solver.

When (Φ, s) fails to output a core directly, we perform Mariposa-style mutations to the Φ, attempting to obtain a core from any of the mutants. We then map the core from a successful mutant back to a core of Φ. If necessary, we also try the core export using different versions of the solver.

Incomplete Core. The solver might also produce a Φ_C that is "incomplete". Specifically, the solver s might return `unsat` on the original query and successfully produce a Φ_C; however, when given Φ_C as input, s fails to produce `unsat`, even with mutants of Φ_C. This could be due to certain assertions that are necessary to the proof but missing in Φ_C. 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 incomplete core to occur more often with F* queries (~8%), and the core is typically only "missing" a small number (≤ 5) of assertions.

To summarize, we make a best-effort attempt to find Φ_C, such that $\Gamma_{\Phi_C} \subseteq \Gamma_\Phi$ and Φ_C is sufficient for some solver s 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 Φ as the Φ_C.

6.2.1.2 Quantifying the Context Relevance

After acquiring an unsatisfiable core, we compare its context to the original. Using the assertion count as a proxy for the "size" of the context, we introduce a metric to quantify the relevance of the context.

Metric: Relevance Ratio. We define the relevance ratio of a query Φ as the proportion of the context that is retained in the core:

$$\frac{\|\Gamma_{\Phi_C}\|}{\|\Gamma_\Phi\|} \times 100\%$$

Since $\Gamma_{\Phi_C} \subseteq \Gamma_\Phi$, the lower this ratio is, the less context is retained, and the more irrelevant context the original query has.

Fig. 6.7 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 for a typical query in the project, only 0.06% of the context is relevant. In vWasm$_F$,

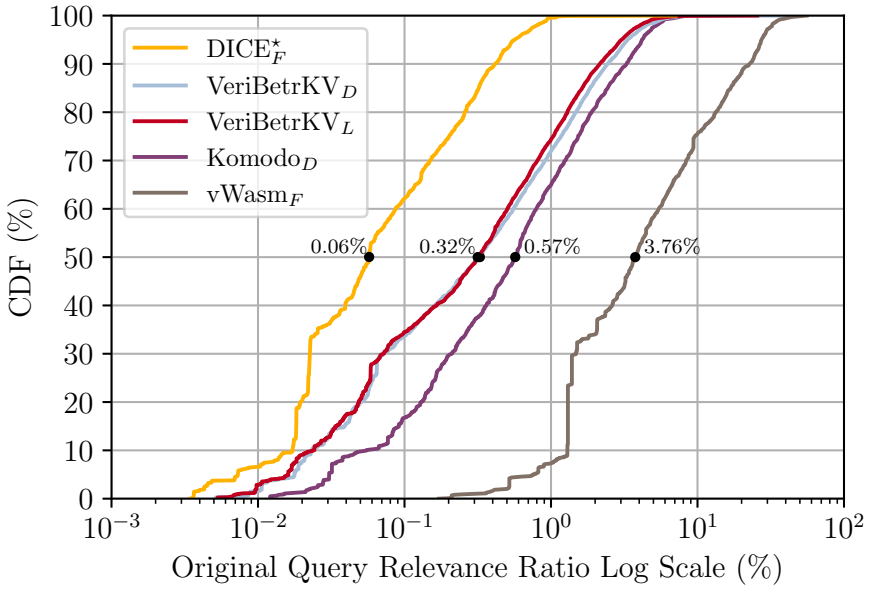


Figure 6.7: **Original Query Context Relevance**

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$_F$, which we discussed in Sec. 6.1.4. Nevertheless, if we consider the complement of the relevance ratio, typically 96.23%–99.94% of the context is irrelevant, even considering vWasm$_F$.

6.2.1.3 Measuring the Stability Impact

Given the significant amount of irrelevant context, we further analyze how that impacts stability. Here we compare and contrast the Mariposa stability status of the original queries and their core counterparts. More generally, for an original query Φ and its counterpart $\Phi' = T(\Phi)$, where T is some mitigation technique, we define the following two metrics to quantify the stability impact:

Metric: Preservation. Given that Φ is stable, the probability that Φ' remains stable.

Metric: Mitigation. Given that Φ is unstable, the probability that Φ' becomes stable.

In this experiment, Φ' is the the core query of Φ, i.e., $\Phi' = \Phi_C$. We use the Mariposa tool with Z3 version 4.12.5 in this experiment. In Fig. 6.8, we list the number of original queries and the scores for solver-produced core queries. 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.

Generally, the solver-produced core is highly likely to preserve originally stable instances.

Project	Original		Core	
	Stable	Unstable	Preservation	Mitigation
$\text{DICE}^{\star}_{F}$	1,483	20	99.6%	90.0%
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%
vWasm_{F}	1,731	4	99.7%	0.0%
Overall	15,110	545	99.5%	78.3%

Figure 6.8: **Stability of Solver-Produced Core**

Moreover, across all projects, 78.3% of the unstable instances can be mitigated with the core. In other words, irrelevant context is **a major contributor to instability**. This result suggests a promising mitigation strategy of pruning irrelevant assertions.

6.2.2 Dissecting the Query Context

We now investigate the composition of query context. In Sec. 2.3.1, we offered an overview of verification condition generation (VCG) in APV. Here we give an intuitive perspective on the verification query as a theorem-proving task.

As we presented in Sec. 2.4, the standard SMT semantics of Φ is the satisfiability, or equivalently, the validity of the entailment $\Gamma_{\Phi} \vdash \bot$. If we consider the axioms separately, $\Gamma_{\Phi} \vdash \bot$ is logically equivalent to $\Lambda_{\Phi} \vdash \theta$. Intuitively, this is a theorem-proving task, where Λ_{Φ} is given to establish the verification goal θ.

As we have demonstrated in Sec. 6.2.1.2, a large portion of Λ_{Φ} might be irrelevant towards θ. Intuitively, the VCG should ensure the completeness of Λ_{Φ}, so that it is theoretically possible to prove θ. Otherwise, there is no way the solver can succeed! Meanwhile, the minimality of Λ_{Φ}, which is in conflict with completeness to some extent, is a secondary concern.

Therefore, the SMT solver is implicitly tasked with *axiom selection* [82], where it needs to choose a subset of the axioms to prove the goal. Using our notation, we can represent the axiom selection task as finding $\Lambda_{\Phi^*} \subseteq \Lambda_{\Phi}$ such that $\Lambda_{\Phi^*} \vdash \theta$. This effectively creates a new query Φ^*, with the reduced context $\Gamma_{\Phi^*} = \Lambda_{\Phi^*} \wedge \neg\theta$,

Connection to Information Retrieval. It is worth noting that axiom selection has a strong analogy to information retrieval (IR) [147]. In IR, we are given a set of documents and a query (e.g., a set of keywords), and the objective is to retrieve a subset of documents that are most relevant to the query.

With the intuitive connection between information retrieval (IR) and axiom selection, IR techniques also offer promising avenues for addressing context pruning challenges. For

instance, in Sec. 6.2.3.3, we demonstrate how the concept of TF-IDF [142] can be adapted to refine the relevance of axioms. Similarly, in Sec. 6.2.3.4, we explore feedback-driven mechanisms inspired by IR to enhance the pruning process. These connections underscore the potential for cross-disciplinary innovation, leveraging IR methodologies to advance APV's efficiency and stability.

6.2.3 Approximating the Relevance

In this section, we introduce SHAKE, a pruning technique APV queries. 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, for a query Φ with $\Gamma_\Phi = \{\psi_0, ..., \psi_n\}$, assuming $\psi_0 = \neg\theta$ encodes the goal, the output of SHAKE is a map of distances:

$$\textsc{Shake}(\Gamma_\Phi) = \{\psi_0 \mapsto 0, ..., \psi_n \mapsto d_n\}$$

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.

6.2.3.1 Leveraging Symbol-Based Relevance

In this version of SHAKE, we abstract a formula ψ via the set of query-defined symbols it contains, denoted as $\textsc{Symbols}(\psi)$. More precisely, the symbols are the functions, constants, and data-types 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. 3 shows the naive SHAKE algorithm. We first initialize a context symbol set Σ_{ctx} from the goal. We then select all axioms ψ_i such that $\textsc{Symbols}(\psi_i)$ intersects with Σ_{ctx}, on the theory that intersection conveys relevance. After scanning through all axioms in this round, we augment Σ_{ctx} with the symbols from the selected axioms. The update is delayed until the end of the round, so Σ_{ctx} remains the same during this scan. Otherwise, the scan order would affect the content of Σ_{ctx}, introducing a form of instability.

Applying this process repeatedly scores the distance of an axiom ψ_i based on the round in which $\textsc{Symbols}(\psi_i)$ first intersects with Σ_{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 Σ_{ctx}, ending the process quickly. In light of this problem, we refine the formula abstraction to handle quantified formulas in a lazy manner.

Algorithm 3 Naive SHAKE

```
procedure NaiveShake(Γ_Φ = {ψ_0, ..., ψ_n})
    # assuming ψ_0 is the goal
    Σ_ctx ← Symbols(ψ_0)
    dists, round ← {ψ_0 ↦ 0}, 1
    repeat
        Σ_acc ← ∅
        for ψ_i ∈ {ψ_1, ..., ψ_n} do
            if Σ_ctx ∩ Symbols(ψ_i) ≠ ∅ then
                # check if ψ_i has been assigned a distance
                if ψ_i ∈ Unreached(dists, Γ_Φ) then
                    dists ← dists ∪ {ψ_i ↦ round}
                end if
                Σ_acc ← Σ_acc ∪ Symbols(ψ_i)
            end if
        end for
        # update the symbol set after considering all ψ_i
        Σ_ctx ← Σ_acc ∪ Σ_ctx
        round ← round + 1
    until IsFixedPoint(dists)
    round ← round + 1
    for ψ_i ∈ Unreached(dists, Γ_Φ) do
        # assign maximum distance to unreachable axioms
        dists ← dists ∪ {ψ_i ↦ round}
    end for
    return dists
end procedure
```

6.2.3.2 Handling Quantified Axioms

As mentioned in Sec. 2.4, pattern-based quantifier instantiation plays an important role in APV languages such as Dafny, F*, and Verus. In this version of SHAKE, we use the available patterns to refine the notion of relevance for the formulas.

We construct a **formula state** for a given formula ψ. We denote this via INITFSTATE(ψ), which augments ψ with two fields:

- $\psi.visible$: the set of symbols in ψ not under quantification.
- $\psi.qstates$: a list of **quantifier states**, constructed only from the outermost quantifiers in ψ. The construction via INITQSTATE is lazy, meaning that any nested quantified formulas are hidden under the outermost quantifier states.

Given a quantified formula ϕ, INITQSTATE(ϕ) creates a quantifier state containing ϕ and two additional fields:

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

SHAKE is lazy when determining the relevance of a quantifier state, reflected in the TRYEXPAND procedure. Given a symbol set Σ_{ctx}, if none of the $\phi.patterns$ is a subset of Σ_{ctx}, the quantified formula is irrelevant, and $\phi.hidden$ 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 Σ_{ctx}. Upon a match, Σ_{ctx} creates a new formula state from its hidden body $\phi.hidden$. We note we only expanded one level of quantifier nesting via INITFSTATE.

```
procedure TRYEXPAND(φ, Σ_ctx)
    relevant ← false
    # subset check needed to check for pattern match
    for Σ_p ∈ φ.patterns do
        if Σ_p ⊆ Σ_ctx then
            relevant ← true
        end if
    end for
    if relevant then
        # create a new formula state from the hidden body
        ϕ ← φ.hidden
        INITFSTATE(ϕ)
        return SOME(ϕ)
    end if
    return NONE
end procedure
```

SHAKE checks the relevance of a formula state ψ as follows. Given a symbol set Σ, ψ

is relevant if $\psi.visible$ intersects with Σ, or if any of the $\psi.qstates$ is considered relevant. When SHAKE expands a quantifier state, the resultant formula state is merged into ψ. This process is reflected in the FORMULARELEVANT procedure below.

```
procedure FormulaRelevant(ψ, Σ_ctx)
    qstates' ← ⟨⟩
    relevant ← Σ_ctx ∩ ψ.visible ≠ ∅
    for φ ∈ ψ.qstates do
        r ← TryExpand(φ, Σ_ctx)
        # expansion may create a new formula state hidden
        if Some(φ) = r then
            # a trigger matches; merge the previously hidden body
            qstates' ← qstates' + φ.qstates
            ψ.visible ← ψ.visible ∪ φ.visible
            relevant ← true
        else
            # no match; no new formula state created
            # append the quantifier state without expansion
            qstates' ← qstates' + ⟨φ⟩
        end if
    end for
    ψ.qstates ← qstates'
    return relevant
end procedure
```

The main procedure for this version of SHAKE is shown in Alg. 4. 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 context, including the goal, in each round, as nested quantifiers may be expanded in later rounds.

6.2.3.3 Handling Frequent Symbols

Thus far we have used the symbol set abstraction introduced in Sec. 6.2.3.1. 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.

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.

Metric: Symbol Frequency. To address this issue, we propose a simple heuristic.

Algorithm 4 Refined SHAKE with Quantifier Patterns

```
procedure SHAKE(Γ_Φ = {ψ_0, ..., ψ_n})
    for ψ_i ∈ Γ_Φ do
        # create the formula state
        INITFSTATE(ψ_i)
    end for
    # assuming ψ_0 is the goal
    Σ_ctx ← ψ_0.visible
    dists, round ← {ψ_0 ↦ 0}, 1
    repeat
        Σ_acc ← ∅
        for ψ_i ∈ Γ_Φ do
            S_prev ← ψ_i.visible
            # possibly expand quantified formulas
            if FORMULARELEVANT(ψ_i, Σ_ctx) then
                if ψ_i ∈ UNREACHED(dists, Γ_Φ) then
                    dists ← dists ∪ {ψ_i ↦ round}
                end if
                # update with previous symbols in ψ_i
                Σ_acc ← Σ_acc ∪ S_prev
            end if
        end for
        Σ_ctx ← Σ_acc ∪ Σ_ctx
        round ← round + 1
    until ISFIXEDPOINT(dists)
    round ← round + 1
    for ψ_i ∈ UNREACHED(dists, Γ) do
        dists ← dists ∪ {ψ_i ↦ round}
    end for
    return dists
end procedure
```

We define the frequency of a symbol f to be the ratio of formulas in $\Gamma_\Phi = \{\psi_0, ..., \psi_n\}$ containing f in their symbol set:

$$freq(f) = \frac{\|\{\psi_i \mid \psi_i \in \Gamma_\Phi \wedge f \in \textsc{Symbols}(\psi_i)\}\|}{\|\Gamma_\Phi\|}$$

Given a threshold ξ, SHAKE excludes all symbols f such that $freq(f) > \xi$, treating them as if they were built-in functions. As a side note, this idea is related to *inverse document frequency* in information retrieval [142]. This simple approach improves pruning on certain F* queries, as we show in the evaluation.

6.2.3.4 Setting Distance Limits

SHAKE is similar to *iterative deepening* [91] 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 ≤ 20, while N can be in the thousands. 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 a solver-produced 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`, which is also a solver configuration. Similar annotations also exist in languages like F* and Verus [94].

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.

6.2.4 Evaluating the Improvement

In this section, we evaluate the effectiveness of SHAKE. We show the distribution of distance values produced by SHAKE in Sec. 6.2.4.1. We then evaluate SHAKE's improvement of context relevance in Sec. 6.2.4.2 and stability in Sec. 6.2.4.3. We further assess the impact of ignoring frequent symbols in Sec. 6.2.4.4. Lastly, in Sec. 6.2.4.5, we evaluate SHAKE's impact on solving performance in terms of run time and number of queries solved.

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 Sec. 6.2.3.
- **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 Sec. 6.2.3.4, 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 (Sec. 6.2.3.3). We only experiment with frequency configuration in Sec. 6.2.4.4.

6.2.4.1 Distribution of Shake Distances

First, we evaluate how well SHAKE distances reflect the relevance of axioms. For a query Φ with context $\Gamma_\Phi = \{\psi_0, ..., \psi_n\}$, SHAKE computes the distances:

$$\text{SHAKE}(\Gamma_\Phi) = \{(\psi_0 : d_0), ..., (\psi_n : d_n)\}$$

Let Γ_{Φ_C} be the context from the solver-produced core. We can then calculate the maximum distances for the original query and the core:

$$d_{orig} = \max(d_i \mid (\psi_i : d_i) \in \text{SHAKE}(\Gamma_\Phi))$$
$$d_{core} = \max(d_i \mid (\psi_i : d_i) \in \text{SHAKE}(\Gamma_{\Phi_C}))$$

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 the layers in between with no loss of core axioms.

As shown in Fig. 6.9-Fig. 6.13, 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, Fig. 6.9 shows the distributions from Komodo$_D$.

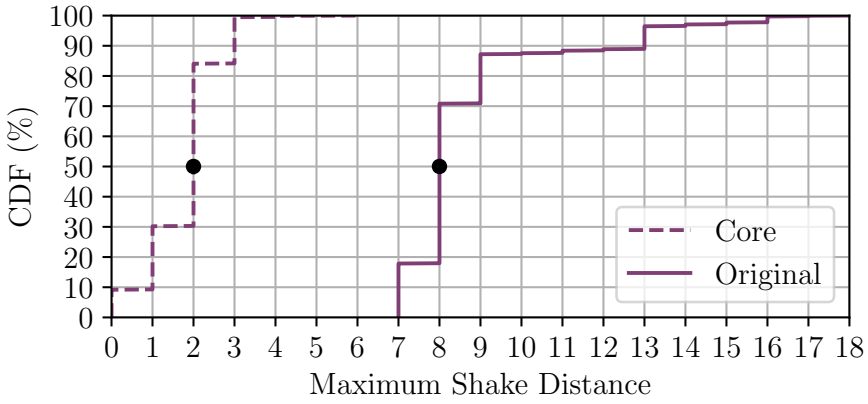


Figure 6.9: **Maximum Shake Distances for Komodo$_D$**

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.

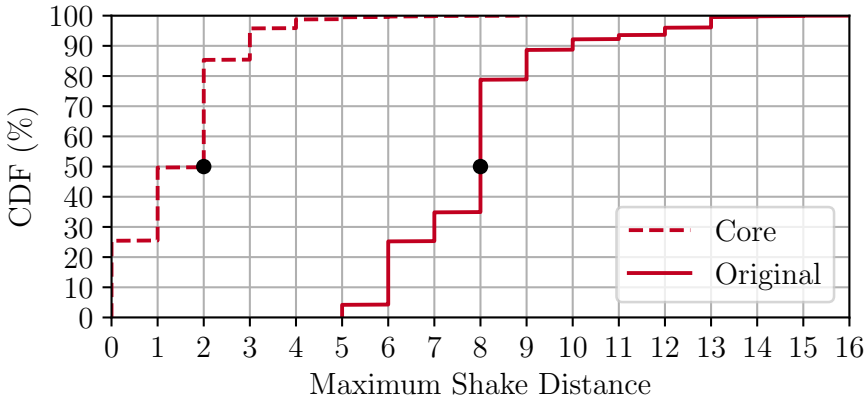


Figure 6.10: **Maximum Shake Distances for VeriBetrKV$_L$**

However, in Fig. 6.13, we observe that vWasm$_F$ is a bit of an outlier (again). As we discussed in Sec. 6.2.1.3, 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.

6.2.4.2 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 Sec. 6.2.1.2, we compute the relevance ratio of the pruned query.

In Fig. 6.14, we present the relevance ratios that SHAKE achieves. We see significant improvements over the original queries as shown in Fig. 6.7. For example, in VeriBetrKV$_L$, the median relevance ratio (MRR) is 0.32% in the original queries, while the MRR increases

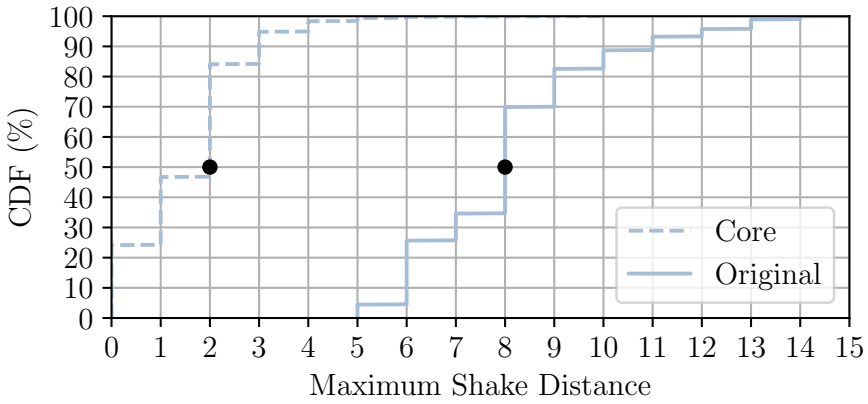


Figure 6.11: **Maximum Shake Distances for VeriBetrKV$_D$**

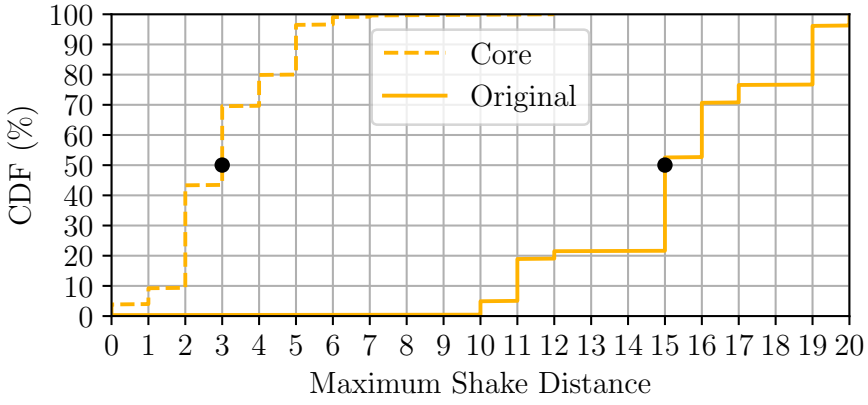


Figure 6.12: **Maximum Shake Distances for DICE$^\star_F$**

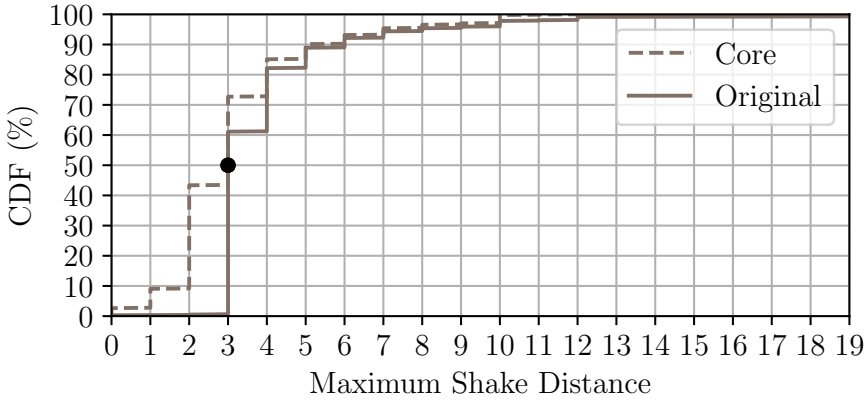


Figure 6.13: **Maximum Shake Distances for vWasm$_F$**

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.

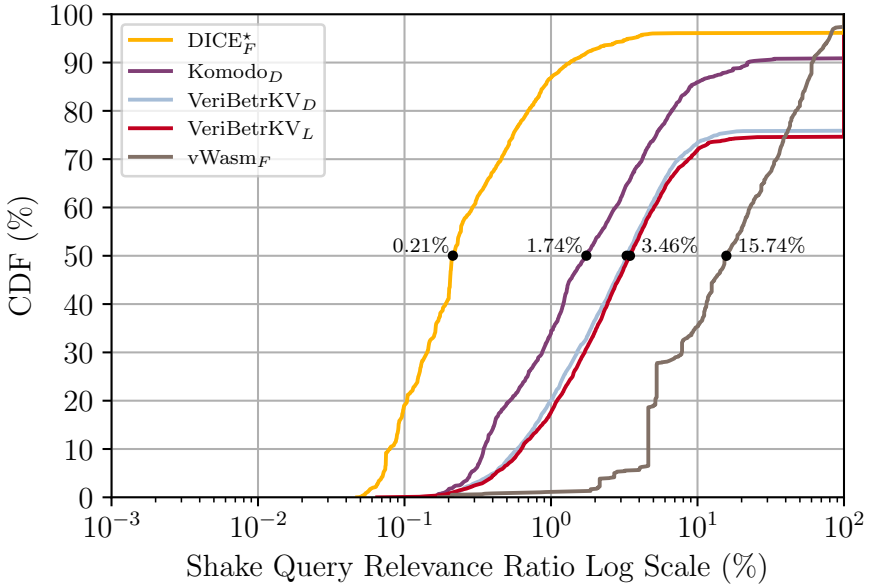


Figure 6.14: **Oracle Shake Query Context Relevance**

6.2.4.3 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 Sec. 6.2.1.3, both from a preservation and mitigation perspective.

In Tab. 6.6, 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 Fig. 6.8), 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 Sec. 6.2.3.1 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 Sec. 6.2.4.4, 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 evaluate Shake, we also run it on cvc5 version 1.1.1. However, we start with the caveat that the cvc5 results are not directly comparable to those from Z3. As we discussed previously, cvc5 cannot solve a significant number of queries from F* and

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%
$\text{DICE}^\star_F$	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%

Table 6.6: **Oracle Shake Stability on Z3 4.12.5**

Dafny, even after we converted them into standard SMT-LIB format, whereas Z3 succeeds for nearly all of them. To bound the total time budget, we only evaluate the stability of original queries that do not timeout with cvc5. This necessarily introduces bias in results, where the given portion of `unstable` queries for cvc5 is a very conservative underestimate.

With that caveat in mind, we present the stability scores for oracle Shake on cvc5 in Tab. 6.7. 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%
$\text{DICE}^\star_F$	110	17	100.0%	100.0%
vWasm_F	110	27	99.1%	0.0%
Overall	550	433	98.7%	41.3%

Table 6.7: **Oracle Shake Stability on cvc5 1.1.1**

6.2.4.4 Frequency Configuration

As discussed in Sec. 6.2.3.3, Shake can optionally take in a threshold ξ and ignore any symbol x such that $freq(x) > \xi$. We now evaluate if this configuration can help with stability. Intuitively, if ξ is set properly, Shake can ignore trivial matches due to pervasively used symbols. However, if ξ 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 ξ with an experiment on query relevance. $\xi = 1.00$ means no symbols are pruned based on frequency. In Tab. 6.8, we observe that there is a trade-off between the relevance ratio and the fallback rate. For example, in Komodo_D, $\xi = 0.15$ achieves the highest MRR, but also has the highest FR. In vWasm_F, since the context starts with high MRR, lower ξ values only increase FR. In general, $\xi = 1.00$ (no frequency pruning) tends to balance the two metrics.

		Orig.	$\xi = 1.00$	$\xi = 0.30$	$\xi = 0.15$
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
$\text{DICE}^\star_F$	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

Table 6.8: **Oracle Shake Context Relevance with Frequency**

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

6.2.4.5 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.** We directly pass the original queries to the solver.
- **Default Shake.** We preprocess the queries Shake in default mode and then pass them to the solver.

- **Oracle Shake.** We preprocess the queries SHAKE in oracle mode and then pass them 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 Tab. 6.9. 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 $\text{DICE}^{\star}_{F}$, 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	1,983	-0.10%	+0.30%
	cvc5	342	+1.75%	+21.64%
VeriBetrKV_{D}	Z3	5,103	-0.78%	-0.61%
	cvc5	2,571	+9.14%	+20.77%
VeriBetrKV_{L}	Z3	5,167	-0.41%	-0.04%
	cvc5	3158	+8.90%	+13.01%
$\text{DICE}^{\star}_{F}$	Z3	1,493	-0.07%	+0.33%
	cvc5	259	+73.36%	+163.71%
vWasm_{F}	Z3	1,733	-0.29%	-0.35%
	cvc5	1,630	-0.12%	-0.12%
Overall	Z3	15,479	-0.45%	-0.18%
	cvc5	7,960	+8.92%	+18.10%

Table 6.9: **Queries Solved with Shake as a Preprocessor**

To present the runtime performance, we use survival plots [28]. 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 Fig. 6.16, 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 Fig. 6.17, VeriBetrKV_{L} shows a similar trend as in VeriBetrKV_{D}.

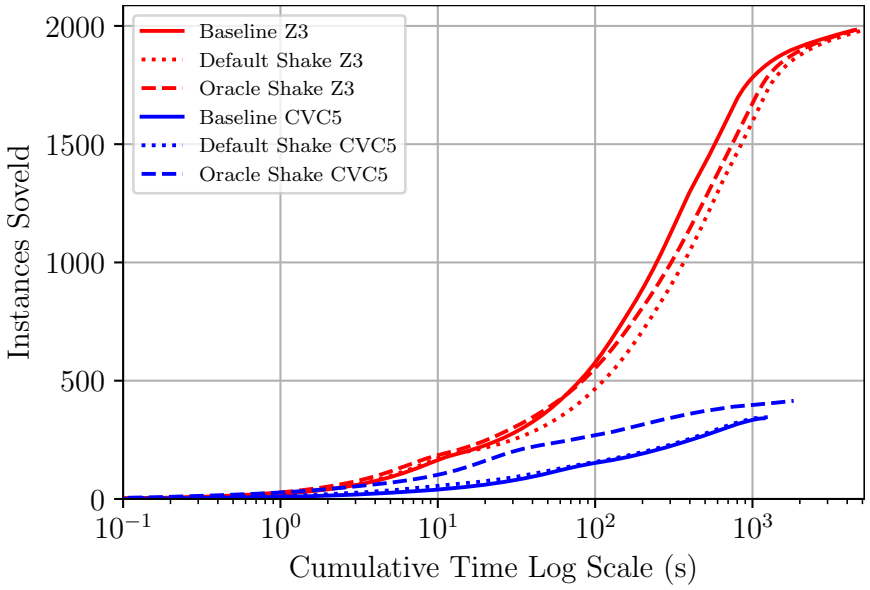


Figure 6.15: **Shake Performance Survival Plot for Komodo$_D$**

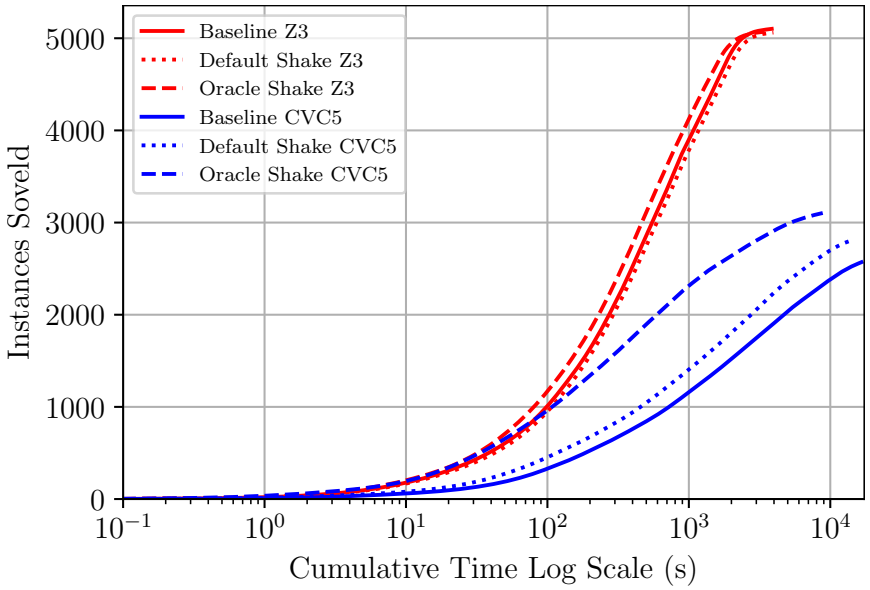


Figure 6.16: **Shake Performance Survival Plot for VeriBetrKV$_D$**

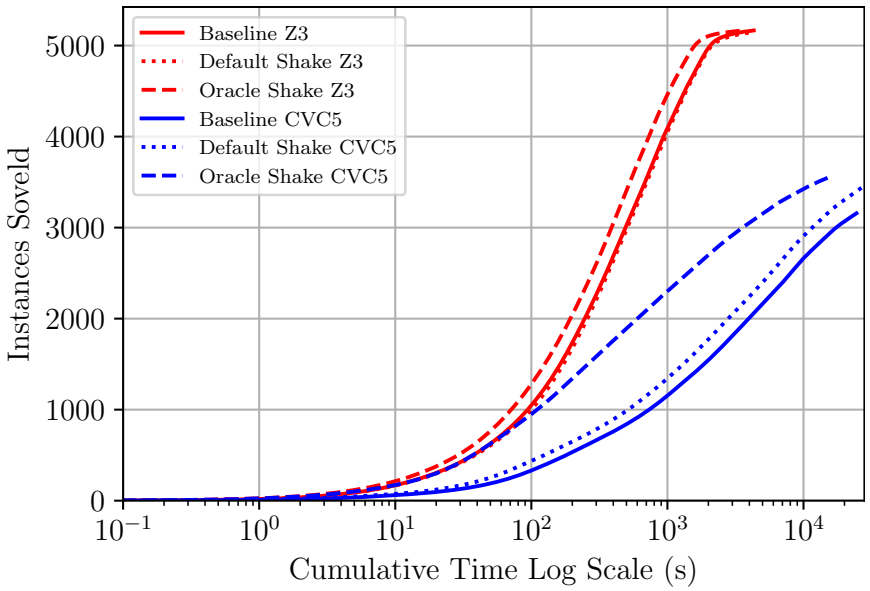


Figure 6.17: **Shake Performance Survival Plot for VeriBetrKV$_L$**

In Fig. 6.17, we show the survival plot for VeriBetrKV$_L$. We observe a similar trend as in VeriBetrKV$_D$. Shake does not have a significant impact on Z3, but helps with cvc5's performance in both modes.

In Fig. 6.18, 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.

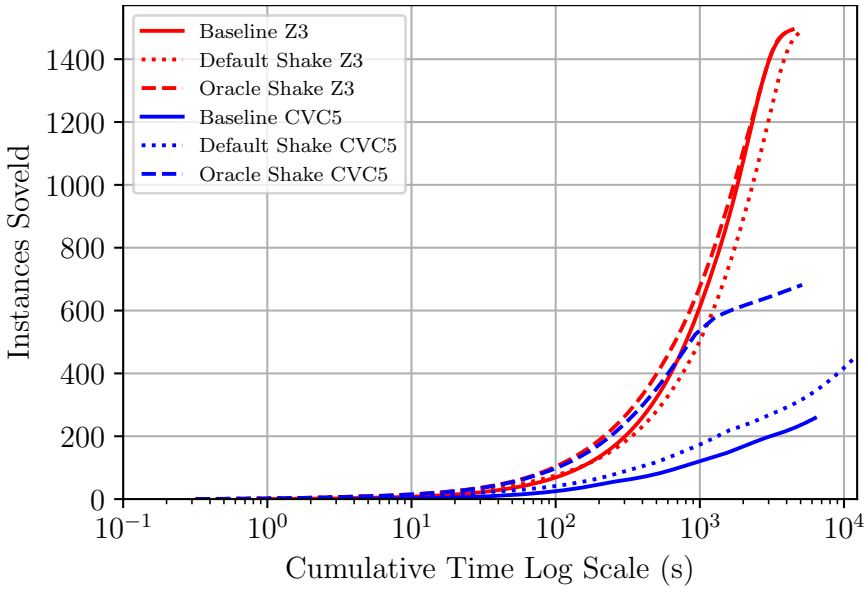


Figure 6.18: **Shake Performance Survival Plot for DICE$^{\star}_{F}$**

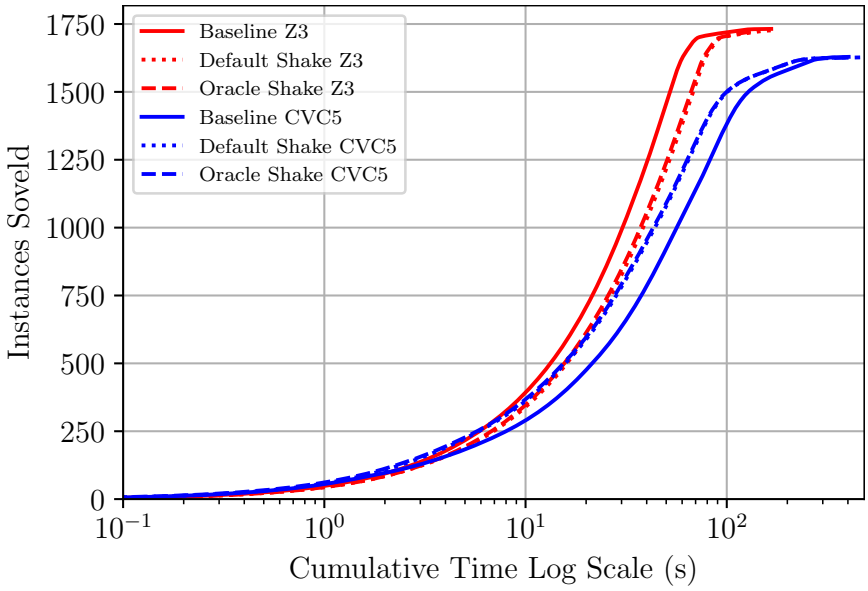


Figure 6.19: **Shake Performance Survival Plot for vWasm$_{F}$**

6.3 Repairing Instability with Cazamariposas

Existing techniques on instability mitigation are often *preventive*. For instance, in default Shake, we attempt to deter stability-related failures at preprocessing time, without prior knowledge of the query. Meanwhile, in oracle Shake, we explore a more *reactive* approach, where we harness solver feedback (i.e., the core) to stabilize future iterations of the query. In this section, we apply the idea to our Cazamariposas tool, which *repairs* unstable APV queries leveraging the solver-constructed proofs.

Conceptually, Cazamariposas searches for *query edits* that repair problematic quantified axioms. We base the edits on the proof, so that an edit **(1)** weakens a single target axiom, and **(2)** preserves the rest of the query context. In this way, a stabilizing edit is also a valid repair strategy, which we envision a developer can reenact at the source level.

For instance, Cazamariposas might identify a quantified axiom ϕ in an unstable query Φ, such that dropping ϕ creates a stable query. Intuitively, $\Gamma_{\Phi^*} = \Gamma_{\Phi} \setminus \{\phi\}$ is stable, and thus ϕ is the cause of instability. Moreover, since Φ^* preserves the verification goal in Φ, the removal of ϕ is also a valid repair strategy, which corresponds to a developer suppressing the axiom at the source level. However, it is worth noting that we assume the translation from query edits to source edits is possible.

In Sec. 6.3.1, we detail our methodology to perform such query edits based on the proof logs. In fact, as our later experiments suggest, there is a *single* problematic axiom to blame in ~61% cases. That is, by limiting the impact of one ill-behaved axiom within a query, we can repair most of the unstable instances and avoid future failures.

However, the vast number of quantified axioms in a query presents a challenge to the repair process. We therefore also discuss how to effectively identify the likely suspects, rather than exhaustively testing all axioms in a query. In Sec. 6.3.2, we make an observation on two distinctive failure modes in which instability manifests: the solver either **(1)** gives up quickly reporting `unknown`, corresponding to under-instantiated axioms, or **(2)** uses up time limit returning `timeout`, corresponding to over-instantiated axioms.

With the triage, we further narrow down the search space of the problematic axioms. We take advantage of the fact that an unstable query Φ has at least a passing mutant Φ_s and a failing mutant Φ_f. In Sec. 6.3.3, we propose metrics based on the instantiation profiles of Φ_s and Φ_f, highlighting quantified formulas that are either over-instantiated or under-instantiated in Φ_f. In Sec. 6.3.4, we further refine the analysis based on the failure mode with novel proof and trace mining techniques, exploiting the causal relation between the instantiations.

6.3.1 Testing the Axioms for Stability

We start with our methodology to evaluate the stability impact of individual axioms, leveraging the information from the proof log. Conceptually, we follow an "edit-and-test"

approach. With a (quantified) assertion $\phi_i \in \Gamma_\Phi$ in an unstable query Φ, we go through the following:

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

More formally, we define a *singleton edit* as a pair (ϕ_i, a_i), where $\phi_i \in \Gamma_\Phi$ is an assertion, and a_i is an *action* among $\{\textbf{del}, \textbf{inst}, \textbf{inst-del}, \textbf{sk}\}$. In Tab. 6.10, we define $\textsc{ApplySingleEdit}(\Gamma_\Phi, \phi_i, a_i)$ as a function that outputs an edited context Γ_{Φ^*} of the candidate query. In particular, for the actions **inst** and **ins-del**, we leverage the instantiation set $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ provided by the proof log.

Action	Applicability	$\textsc{ApplySingleEdit}(\Gamma_\Phi, \phi_i, a_i)$
del	$\phi_i \in \Lambda_\Phi, \textsc{IsQuant}(\phi_i)$	$\Gamma_\Phi \setminus \{\phi_i\}$
inst	$\phi_i \in \Lambda_\Phi, \textsc{IsForall}(\phi_i)$	$\Gamma_\Phi \cup \mathcal{I}_{\mathfrak{p}}^{\phi_i}$
inst-del		$(\Gamma_\Phi \cup \mathcal{I}_{\mathfrak{p}}^{\phi_i}) \setminus \{\phi_i\}$
sk	$\phi_i \in \Gamma_\Phi, \phi_i = \exists x.\varphi$	$(\Gamma_\Phi \cup \{\varphi[x \mapsto f_{\phi_{ix}}]\}) \setminus \{\phi_i\}$

Table 6.10: **Cazamariposas Query Edits**

Intuitively, the edits are meant to reduce or eliminate the reasoning obligation ϕ_i introduces, so a stabilized candidate also points to ϕ_i as a cause of instability. We now discuss some basic properties of the $\textsc{ApplySingleEdit}$, including soundness, which ensures that a stabilizing edit is also a valid repair strategy.

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

$$\Gamma_{\Phi^*} \vdash \bot \implies \Gamma_\Phi \vdash \bot$$

We can demonstrate the soundness with a case analysis. When $a_i = \textbf{sk}$, $\textsc{ApplySingleEdit}$ leaves the query semantics unchanged, so soundness trivially holds. For the rest of the actions, the goal θ remains unchanged. (Note that we have restricted **sk** to be the only potential action on goal.) Therefore, we could instead show that:

$$\Lambda_{\Phi^*} \vdash \theta \implies \Lambda_\Phi \vdash \theta$$

which holds as long as Λ_{Φ^*} is no stronger than Λ_Φ. When $a_i = \textbf{inst}$, because the elements of $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ are tautological consequences of ϕ_i, Λ_{Φ^*} is as strong as Λ_Φ. When $a_i \in \{\textbf{del}, \textbf{inst-del}\}$, Λ_{Φ^*} might be weaker than Λ_Φ. Therefore, soundness also holds for the rest of the actions.

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

$$\Lambda_\Phi \vdash \theta \implies \Lambda_{\Phi^*} \vdash \theta$$

The proof instantiation set $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ is sufficient[2] to establish θ, so we maintain completeness by this definition. However, since the edits may weaken the axioms, we do sacrifice a broader sense of completeness. Specifically, **del** and **inst-del** may remove a quantified axiom ϕ_i, while the $\mathcal{I}_{\mathfrak{p}}^{\phi_i}$ is only a finite subset of all possible instantiations of ϕ_i, and thus we can no longer guarantee that $\Lambda_{\Phi^*} \vdash \phi_i$.

Applicability. As shown in Tab. 6.10, we limit the actions to quantified axioms, with the only exception of $(\varphi_0, \mathbf{sk})$. As a result, we can establish soundness relatively easily, but we might miss out certain problematic axioms as candidates. For example, consider an axiom $\phi_i = (\forall x.\varphi_i) \vee (\forall x.\varphi_j)$. Even though ϕ_i contains quantified formulas, we do not target it for the stability test. (Soundness arguments become a lot more subtle if we were to allow edits on more general axioms with quantified sub-formulas, rather than just quantified axioms.) This is a limitation of our current approach.

Composability. More generally, we define the function:

$$\textsc{ApplyEdits}(\Gamma_\Phi, \Delta = \langle ..., (\phi_i, a_i), ... \rangle)$$

where Δ is a sequence of singleton edits. ApplyEdits performs the edits in Δ sequentially, while maintaining soundness and completeness. Intuitively, when instability arises from the interaction of multiple quantified axioms, singleton edits (i.e., $\|\Delta\| = 1$) might fall short to capture the cause, and thus we need to consider $\|\Delta\| \geq 2$. In the case where $\|\Delta\| = 2$, we call Δ a *doubleton edit*.

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

- **del.** As we discussed in Sec. 2.3.1, APV languages typically offers source-level visibility control mechanisms. For example, Dafny's `opaque` keyword allows developers to hide the definition of a function.
- **inst.** The developer can also directly introduce quantifier instantiations as source-level annotations. For example, Line 3 in Lst. 2.1 explicitly adds `(a+b)*c == a*c + b*c` into the solver's context, which is an instantiation of the distributive property.
- **sk.** APV languages usually has Hilbert's choice as a language construct. For example, Dafny's `var x :| P(x)` assigns `x` an arbitrary value such that `P(x)` holds.

However, the translation might not always be straightforward. As we discussed in Sec. 2.3.1, the axioms may also encode the semantics of language constructs. For example, **del** on an axiom for higher-order functions has no direct source-level equivalent. More specific to **inst**, if the repair adds a large number of instantiations to the source code, it is arguably impractical due to maintenance and readability concerns. Nevertheless, we have some empirical evidence that the repairs are often practical, which would make it interesting to explore automatic translation in the future, potentially through ProofPlumber (Sec. 4.2).

[2] Otherwise, how did the proof succeed in the first place?

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

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

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

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

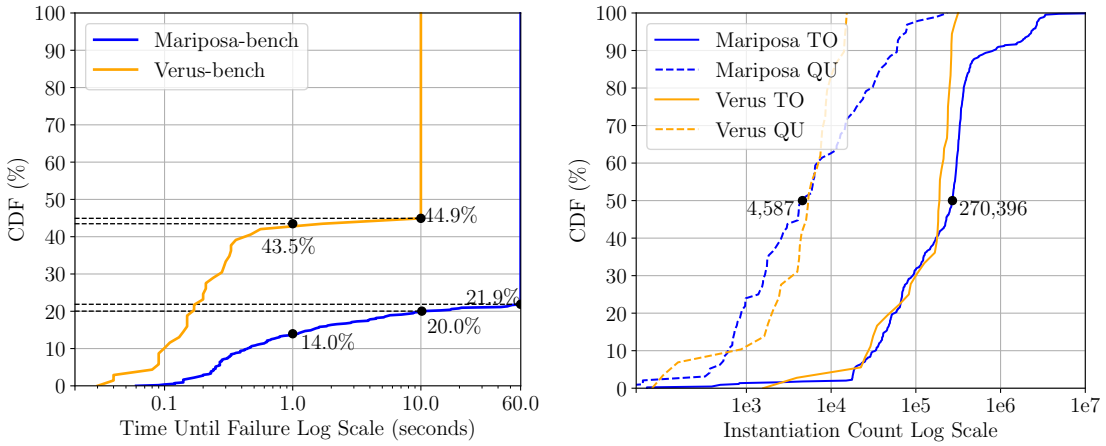
6.3.2 Triaging the Failure Modes

To help narrow down the search space of the problematic axioms, we make an observation on the failure modes of the unstable queries. That is, for an unstable query Φ, how the failed member(s) of the sample mutants $\hat{M}_\Phi$ behave. More specifically, we pick an arbitrary failed mutant $\Phi_f \in \hat{M}_\Phi$. We observe two distinct failure modes of Φ_f, which we name *quick unknown* (QU) and *slow timeout* (TO). In a QU, the solver quickly terminates with an `unknown` result, despite being given a generous timeout and resource limit. Meanwhile, in a TO, the solver runs on the query until it runs out of its time or resource budget.

For this experiment, we use both the Mariposa Bench and the Verus Bench (Sec. 6.1.4). As we mentioned previously, we set a solver time limit of 60 seconds for the former, and 10 seconds for the latter. In Fig. 6.20a, for each original query Φ in the benchmarks, we report the runtime of Φ_f. The plot is in log scale, and we observe that the distribution for each benchmark is bimodal. For Verus Bench, ~43% of the failures occur within 1 second, barely any occur between 1 and 10 seconds, and the rest fail at 10 seconds. For Mariposa Bench, the distribution is more spread out, but the separation is still clear, where ~19% queries fail within 10 seconds, and ~78% time out after 60 seconds. There is almost no middle ground between the two modes.

We perform our triage based on the solver output (i.e., `unknown` or `timeout`), and then plot the instantiation counts based on the failure modes. We note the log-scale on x-axis, highlighting the fact that the TO cases have orders of magnitude more instantiations than the QU ones. For example, in Mariposa TO, the median instantiation count is $270,396$ while in Mariposa QU, the median is $4,587$. The separation is also clear within Verus benchmark.

Our analysis suggests the two failure modes correspond to different types of problems in quantifier reasoning. Specifically, for QU, we hypothesize that certain formulas are



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

Figure 6.20: **Failure Mode Distinction**

insufficiently instantiated in failed runs. For TO, we hypothesize that certain formulas are excessively instantiated, or have a high impact on the instantiations of other quantified formulas.

6.3.3 Calculating the Differential Metrics

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

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

Concretely, we obtain from the solver a trace log $\mathfrak{t}$ for Φ_f, and a proof log $\mathfrak{p}$ for Φ_s. The log files essentially contain *instantiation profiles*, which we introduced in Sec. 2.4. In theory, our method generalizes to multiple traces and proofs. In practice, collecting even one pair of $(\mathfrak{t}, \mathfrak{p})$ can entail difficulties. Hence, we focus our discussion on a pair of proof and trace. Furthermore, since we are comparing the instantiation profiles modulo the structural isomorphism between Φ_f and Φ_s, for the ease of exposition, we omit the detailed subscripts for $\mathfrak{t}$ and $\mathfrak{p}$, which would otherwise be $\mathfrak{t}_{\Phi_f}$ and $\mathfrak{p}_{\Phi_s}$.

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

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

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

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

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

6.3.4 Ranking the Potential Edits

As mentioned in Sec. 6.3.1, we use a parameter k to limit the number of candidates we test for full stability, where we first consider top-k singleton edits, and then doubleton edits if none of the singleton edits is stabilizing. We describe how to compute the scores for the singleton and doubleton edits in this section. The output of this stage are two partial maps, SScore and DScore.

1. We score each axiom ϕ_i, and then select an appropriate edit for it. When there are multiple possible actions on ϕ_i, we commit to one that is likely stabilizing. More formally, we create a partial map:
$$\text{SScore} = \{(\phi_i, a_i) \mapsto s_i \mid \phi_i \in \Gamma_\Phi\}$$
where $a_i \in \{\textbf{del}, \textbf{inst}, \textbf{inst-del}, \textbf{sk}\}$ is the chosen edit action.
2. We then score ordered pair of quantified assertions, along with the most promising actions for each assertion. More formally, we create another partial map:
$$\text{DScore} = \{\langle(\phi_i, a_i), (\phi_d, a_j)\rangle \mapsto s_{ij} \mid \phi_i, \phi_j \in \Gamma_\Phi\}$$

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

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

6.3.4.1 Ranking Edits for QU

We start with how we handle QU failures, which is more straightforward. At the general triage (Sec. 6.3.2) stage, we hypothesize that the QU failures are due to the absence of certain instantiations. Intuitively, we are looking for under-instantiated axioms, where **inst** is applicable, i.e.,

$$\text{SScore} = \{(\phi_i, \textbf{inst}) \mapsto s_i \mid \phi_i \in \Gamma_\Phi\}$$

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

1. (Deficit, −Excess)
2. (−Excess, Deficit)
3. $\kappa \cdot$ Deficit − Excess for some constant κ
4. Deficit/Excess

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

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

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

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

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

6.3.4.2 Ranking Edits for TO

In the general triage stage (Sec. 6.3.2), we hypothesize that the solver is spending significant time and resources on irrelevant quantified formulas in a TO failure. For this failure mode,

we focus on the *quantified axioms* in Λ_Φ as targets. Intuitively, we would need to suppress the excessive instantiation to stabilize the query, while editing the goal is not an option.

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

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

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

Dependency Analysis

In order to locate the root cause of TO instability. we further analyze the causal relations between the instantiations. Our notion of causality extends the *instantiation graph* from the SMTScope (formerly the Axiom Profiler) [16], a tool to analyze instantiation loops and other sources of poor performance in pattern-based SMT solvers. Below, we describe Axiom Profiler's approach and then our extension.

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

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

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

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

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

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

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

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

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

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

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

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

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

The fixed-point computation is non-decreasing by transitivity. However, there is no theoretical guarantee that it will converge. In particular, when G_{ϕ_i} contains a cycle, certain node's ratio may approach a limit at an exponential decay rate. Nevertheless, this is not a

threat to practical usage. In particular, since floating point numbers represent the ratios, the convergence criteria must be threshold-based. For our implementation, we consider a ratio to have converged if its increment from the previous iteration is $\leq 10^{-4}$.

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

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

6.3.5 Evaluating the Improvement

In this section, we evaluate Cazamariposas' ability to automatically identify stabilizing edits. We use a total of 614 unstable queries, with 545 from Mariposa Bench and 69 from Verus Bench, as described in Sec. 6.1.4.1. In our evaluation, we set the number of candidate edits $k = 10$. That is, we let Cazamariposas try $k = 10$ singleton candidates, and if none works, we try 10 doubleton edits.

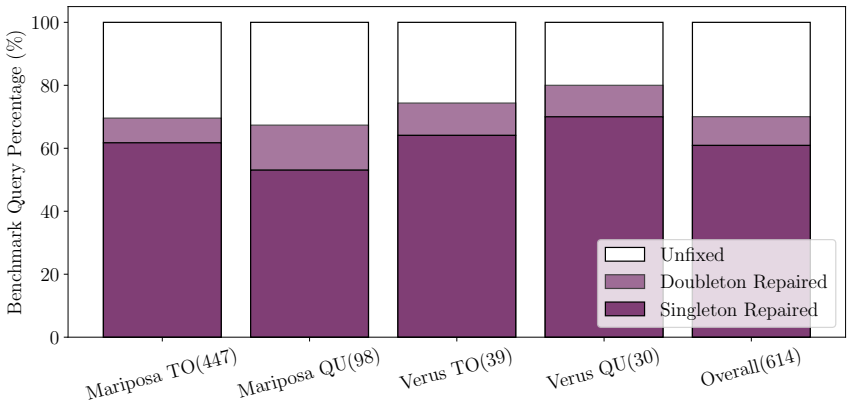
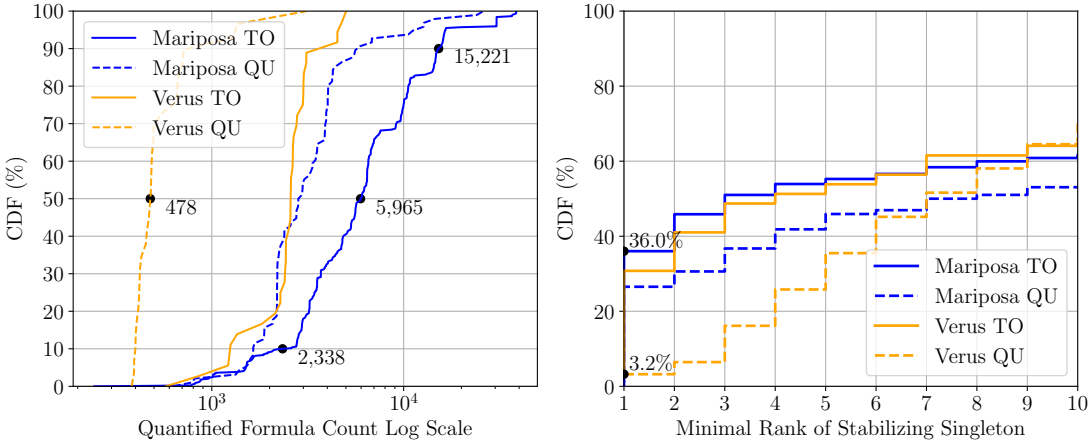


Figure 6.21: **Percentage of Benchmark Queries Repaired**

Overall, Cazamariposas repairs 430/614 ($\approx 70\%$) of the benchmark queries. Fig. 6.21 provides more details, reporting Cazamariposas' performance on the two benchmarks, subdivided by the underlying failure type (TO vs. QU). We note that Mariposa TO accounts for the largest absolute number of queries. Cazamariposas appears to be more effective on Verus queries in either failure type. Nevertheless, Cazamariposas repairs approximately 69% of the Mariposa queries and 77% of the Verus queries. This compares

favorably to the best results from SHAKE (Sec. 6.2.4), which stabilized 29% of the Mariposa benchmark. We also observe that 374/614 ($\approx 61\%$) queries can be stabilized with a single edit. Doubleton edits subsequently provide a small but noticeable boost.



(a) Quantified Formula Count.

(b) Rank of Stabilizing Edits.

Figure 6.22: **Finding Repairs Among Quantified Formulas**

We also evaluate how well Cazamariposas ranks the stabilizing edits. First, in Fig. 6.22a, we show the distribution of the number of quantified formulas in the original queries. For example, in Mariposa TO, the median count is $5,965$, which is a large search space for possible edits. The median count is lower in Verus QU, making it potentially more tractable to fully explore.

Now that we have sense of the search space, we evaluate how well Cazamariposas identifies the useful edits. In Fig. 6.22b, we report the minimal rank of the stabilizing singleton edits. Specifically in singletons, given a query, Cazamariposas produces a ranked list of 10 edits, and we report the rank of the first stabilizing edit within this list. We note the endpoints of the CDFs on the y-axis. It is the probability that Cazamariposas finds a stabilizing edit within the first 10 singleton edits, which corresponds to Fig. 6.21. We note the start points of the CDFs on the y-axis. This is the probability that the first edit Cazamariposas tries would directly work. For a Mariposa TO query, Cazamariposas has a 36% chance of finding a stabilizing edit with one shot.

6.4 Work Status and Personal Contribution

I am the lead author of the three papers in this chapter. In general, I designed, implemented, and experimented with multiple iterations of the methodology, carried out the experiments, and did most of the paper writing. I would like to highlight the contributions of my collaborators in this section.

The work on Mariposa (Sec. 6.1) was published in FMCAD 2023 [173]. Jay Bosamiya had indirect but non-trivial contributions to the design of the Mariposa methodology.

The work on Shake (Sec. 6.2) was published in FMCAD 2024 [172]. I based Shake on an earlier idea from my advisor, Bryan Parno. Jessica Li had non-trivial contributions to experiments on the unsatisfiable core.

The work on Cazamariposas (Sec. 6.3) has just been accepted to CADE 2025 as of the time of writing [175]. The general idea behind the differential analysis came up during discussions with my advisor, Bryan Parno. Amar Shah had a non-trivial contribution to collecting the proof logs, conducting the experiments, and writing the paper. Zhengyao Lin worked on prototyping and validating of our "edit-then-test" approach.

Chapter 7

Conclusion

In this thesis, we have discussed several major scalability challenges in Automated Program Verification (APV) for system software. Let us review the thesis statement:

> *While fully automated program verification is impossible,*
> *we can often have scalable solutions to practical systems,*
> *based on the recurrent reasoning and programming patterns.*

We believe that we have provided sufficient evidence to support this statement, covering various stages of the verification process:

In terms of developing proofs (Chapter 3), we first discussed the challenges of memory reasoning. While the general case aliasing can be very complex, we have shown that in practice, linearity is the common case in system software. We thus introduced linear types to exploit the pattern, not only improving the performance of the verification process, but also reducing the complexity of the specification. We also discussed the challenges theory-specific reasoning, where we converted the existing manual proof patterns into automated encoding in the verification condition generator (VCG).

In terms of debugging proofs (Chapter 4), while fixing a proof failure is intrinsically manual, we again make observations on the common manual patterns. We have shown that these patterns are amenable to mechanization, and we further provided a framework for developers to define their own debug automation.

In terms of reusing proofs (Chapter 5), we demonstrated how to decompose a monolithic system software verification task into smaller, modular, and reusable components. By leveraging appropriate abstractions and language features, such as functors, we achieved a significant reduction in verification effort. Additionally, our curated standard library, built from past projects, highlights the recurrent reasoning patterns prevalent in APV projects.

In terms of stabilizing proofs (Chapter 6), we face yet another severe challenge rooted in the undecidable nature of APV. To tackle this challenge, we developed a systematic methodology to quantify and detect instability, moving from anecdotal observations into

rigorous statistical analysis. Through carefully designed experiments, we identified recurring causes of instability and proposed targeted mitigation strategies. We further leverage the unique structural properties of APV queries in our mitigation, resulting in a substantial improvement in stability and robustness of the verification process.

While we have made non-trivial progress in addressing the scalability problems, there is still much work to be done. Certain aspects of APV remain challenging, and new issues will undoubtedly emerge as systems grow in complexity and scale. However, we are confident that the principles and methodologies outlined in this thesis provide a solid foundation for tackling these challenges. Building on the patterns and insights we have identified, we believe that scalable and practical solutions for system software verification will be within reach.

Bibliography

[1] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1996. 69

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017. 1, 52, 65

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *IEEE Symposium on Security and Privacy*, 2020. 1, 52, 65

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013. 9

[5] Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, April 2015. 52

[6] Arvind Arasu, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Aymeric Fromherz, Kesha Hietala, Bryan Parno, and Ravi 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. 1

[7] arm Holdings plc. Neon. `https://developer.arm.com/Architectures/Neon`, 2022. 52

[8] V. Astrauskas, P. Muller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2019. 9

[9] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based Automated Reasoning for AWS Access Policies Using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018. 1

[10] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms*

for the Construction and Analysis of Systems (TACAS), 2022. 1, 9

[11] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 9

[12] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011. 83

[13] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, 2005. 75

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. www.SMT-LIB.org. 11

[15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. 1, 11

[16] Nils Becker, Peter Müller, and Alexander J Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019. 118

[17] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Security Symposium*, 2015. 52

[18] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. 8

[19] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, July 2021. 81

[20] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. An in-depth symbolic security analysis of the acme standard. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2601–2617, 2021. 1

[21] Lutz Bierl. *MSP430 Family Mixed-signal Microcontroller Application Reports*. Analog and mixed-signal. Texas Instruments, January 2000. 65

[22] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, August 2011. 9

[23] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch,

Bryan Parno, Ashay Rane, Srinath TV Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, volume 152, 2017. 1, 9, 52, 54, 65

[24] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar 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. 1

[25] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. Verified transformations and Hoare logic: Beautiful proofs for ugly assembly language. In *Proceedings of the Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, July 2020. 52, 65

[26] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*, August 2022. 1, 81

[27] Robert S Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979. 8

[28] Martin Brain, James H Davenport, and Alberto Griggio. Benchmarking solvers, SAT-style. In *SC²@ ISSAC*, 2017. 107

[29] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 2007. 8

[30] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 9

[31] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 2008. 9

[32] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17. Association for Computing Machinery, 2017. 81

[33] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021. 54

[34] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere: Towards practical verified kernels in rust. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification (KISV)*, 2023. 81

[35] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 Software.

In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014. 52, 65

[36] Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno. A framework for debugging automated program verification proofs via proof actions. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, July 2024. 3

[37] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. Continuous Formal Verification of Amazon s2n. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2018. 1

[38] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. 81

[39] Byron Cook. Formal reasoning about the security of amazon web services. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, 2018. Springer International Publishing. 1

[40] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. 69

[41] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011. 9

[42] Dafny Contributors. Bit vector cookbook. 37

[43] Dafny Contributors. Dafny Standard Libraries Repository. 3

[44] Dafny Contributors. Verification Debugging When Verification Fails. 43, 48

[45] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 8

[46] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 12

[47] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. 1, 9, 83

[48] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and Proving the TLS 1.3 Record Layer. In *IEEE Symposium on Security and Privacy (SP)*, 2017. 1

[49] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021. 1, 2, 41

[50] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM*, 2022. 9

[51] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, aug 1975. 8, 16

[52] Mike Dodds. Formally Verifying Industry Cryptography. *IEEE Security and Privacy Magazine*, 20(3), 2022. 75

[53] Derek Dreyer. *Understanding and Evolving the Ml Module System*. PhD thesis, Carnegie Mellon University, USA, 2005. 58

[54] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014. 9

[55] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003. 9

[56] Marco Eilers and Peter Müller. Nagini: A Static Verifier for Python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, Cham, 2018. Springer International Publishing. 9

[57] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, 2019. 52

[58] William Feller. *An Introduction to Probability Theory and its Applications, Volume 2*, volume 81. John Wiley & Sons, 1991. 78

[59] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017. 1, 37, 54, 55, 81

[60] Melvin Fitting. *First-order Logic and Automated Theorem Proving*. Springer Science & Business Media, 2012. 91

[61] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 1967. 8

[62] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. `https://falcon-sign.info/`, November 2017. 65

[63] FStar Contributors. Sliding Admit Verification Style. 43

[64] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. In *Proceedings of the National Computer Security Conference*, 1989. 64

[65] Michael JC Gordon. Hol: A proof generating system for higher-order logic. In *VLSI*

specification, verification and synthesis. Springer, 1988. 8

[66] Gert-Martin Greuel, Gerhard Pfister, Olaf Bachmann, Christoph Lossen, and Hans Schönemann. *A Singular introduction to commutative algebra*, volume 348. Springer, 2008. 39, 40

[67] Shay Gueron. Intel Advanced Encryption Standard (AES) New Instructions Set. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, September 2012. 52

[68] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 2011. 9

[69] Arie Gurfinkel and Nikolaj Bjørner. The science, art, and magic of constrained horn clauses. In *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2019. 14

[70] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF 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. 81

[71] Travis Hance. *Verifying Concurrent Systems Code*. PhD thesis, Carnegie Mellon University, August 2024. 37

[72] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan 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. 1, 35, 81

[73] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020. 54

[74] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan 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. 1

[75] David Harel. *First-order dynamic logic*. Springer, 1979. 8

[76] John Harrison. Automating elementary number-theoretic proofs using gröbner bases. In *Proceedings of the Conference on Automated Deduction (CADE)*, 2007. 39

[77] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015. 14

[78] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating*

Systems Principles (SOSP), 2015. 1, 37, 54, 81

[79] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014. 1, 37, 54, 55, 75

[80] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 107–124. IEEE, 2022. 1

[81] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 1969. 8

[82] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *International Conference on Automated Deduction*, pages 299–314. Springer, 2011. 94

[83] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 2001. 51, 52

[84] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, December 2017. 10

[85] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*. Springer, 2016. 9

[86] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. Pqclean. `https://github.com/PQClean/PQClean` Commit febf78a. 72

[87] Richard M Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009. 9

[88] Ioannis T Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Proceedings on the International Symposium on Formal Methods (FM)*, 2006. 26

[89] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-aided reasoning: ACL2 case studies*, volume 4. Springer Science & Business Media, 2013. 8

[90] Matt Kaufmann and J Strother Moore. Acl2: An industrial strength version of nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*, pages 23–34. IEEE, 1996. 8

[91] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 1985. 100

[92] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis*

of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20, pages 389–391. Springer, 2014. 9

[93] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, 2017. 2, 28

[94] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris 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. 3, 9, 37, 41, 100

[95] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1533–1557, 2023. 9

[96] K. R. M. Leino and Clément Pit-Claudel. Trigger Selection Strategies to Stabilize Program Verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2016. 75

[97] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010. 1, 9

[98] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *International School on Foundations of Security Analysis and Design*, pages 195–222. Springer, 2007. 9

[99] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009. 10

[100] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris 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. 1, 2, 31, 41, 81

[101] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of ACM SIGCOMM*, pages 490–503, New York, NY, USA, 2018. Association for Computing Machinery. 1

[102] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Grossschadl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In *Proceedings of IACR Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2015. 69

[103] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Proceedings of the Conference on Cryptology*

and Network Security (CANS), 2016. 69

[104] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2020. 1, 37, 54

[105] Andrey Marochko, Dennis Mattoon, Paul England, Ronald Aigner, Rob Spiger (CELA), and Stefan Thom. Cyber-Resilient Platforms Overview. Technical Report MSR-TR-2017-40, Microsoft Research, September 2017. 81

[106] Joao P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 1999. 9

[107] Y Matiyasevič. Enumerable sets are diophantine. *Mathematical logic in the 20th century*, pages 269–273, 2003. 12

[108] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2021. 9

[109] William McCune. Otter 2.0. In *10th International Conference on Automated Deduction: Kaiserslautern, FRG, July 24–27, 1990 Proceedings 10*, pages 663–664. Springer, 1990. 8

[110] Kenneth L McMillan and Oded Padon. Ivy: a multi-modal verification tool for distributed algorithms. In *Computer Aided Verification (CAV)*, pages 190–202. Springer, 2020. 1

[111] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 146–161. Springer, 2012. 9

[112] Microsoft. Language Server Protocol. 48

[113] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997. 51

[114] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985. 54

[115] Michał Moskal. Programming with triggers. In *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2009. 12, 23

[116] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, 2001. 9

[117] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *28th International Conference on Automated Deduction*. Springer, July 2021. 8

[118] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International*

Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2016, Berlin, Heidelberg, 2016. Springer-Verlag. 9

[119] Jorge A Navas, Bruno Dutertre, and Ian A Mason. Verification of an optimized ntt algorithm. In *Verified Software: Theories, Tools, and Experiments*, 2020. 69

[120] Hamid Nejatollahi, Nikil D. Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. Post-quantum lattice-based cryptography implementations. *ACM Computing Surveys (CSUR)*, 51, 2019. 69

[121] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. 23

[122] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019. 4, 14, 81

[123] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017. 1, 14

[124] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020. 1, 14

[125] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. 8

[126] OpenSSL Developers. OpenSSL. 37

[127] OpenSSL Developers. OpenSSL Vulnerabilities. 52

[128] OpenTitan . OTBN simulator. https://github.com/lowRISC/opentitan/tree/0be5abcf448de4e6076067820e27fbc77bd93a72/hw/ip/otbn/dv/otbnsim. 72

[129] OpenTitan. The OpenTitan Project. https://opentitan.org/. 64

[130] J. M. Owre, S.and Rushby and N. Shankar. Pvs: A prototype verification system. In *Automated Deduction*, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. 8

[131] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017. 14

[132] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016. 9

[133] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011. 64

[134] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively proposi-

tional logic using dpll and substitution sets. *Journal of Automated Reasoning*, 2010. 14

[135] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 1977. 8

[136] John M Pollard. The fast fourier transform in a finite field. *Mathematics of computation*, 1971. 56

[137] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago 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. 1, 19, 52

[138] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Proceedings of the Conference on Concurrency Theory (CONCUR)*, 2018. 52, 65

[139] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy*, 2019. 1

[140] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020. 1, 52, 65

[141] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017. 19

[142] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, 2003. 95, 100

[143] John C Reynolds. Separation logic: A logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. 8, 26

[144] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953. 2

[145] Rust Analyzer Contributors. Rust Analyzer. 48

[146] Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, January 2008. 9

[147] Hinrich Schutze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008. 94

[148] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy*, pages 138–157. IEEE, 2016. 9

[149] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 287–305, Carlsbad, CA, October 2018. USENIX Association. 1

[150] StackOverflow Users. StackOverflow Question: Hint on FStar Proof Dead End. 43

[151] StackOverflow Users. StackOverflow Question: With Dafny, Verify Function to Count Integer Set Elements less than a Threshold. 43

[152] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024. 81

[153] Geoff Sutcliffe and Christian Suttner. The tptp problem library. *Journal of Automated Reasoning*, 21:177–203, 1998. 8

[154] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016. 1, 9

[155] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. DICE*: A Formally Verified Implementation of DICE Measured Boot. In *Proceedings of the USENIX Security Symposium*, August 2021. 1, 81

[156] Aaron Tomb. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security Privacy Magazine*, 14(6), November 2016. 52

[157] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 116, 2011. 64

[158] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017. 39, 52

[159] Verified IronKV. https://github.com/verus-lang/verified-ironkv. Accessed Feb. 2025. 81

[160] Verified memory allocator. https://github.com/verus-lang/verified-memory-allocator. Accessed Feb. 2025. 81

[161] Verified node replication. https://github.com/verus-lang/verified-node-

replication. Accessed Feb. 2025. 81

[162] Verified page table for NrOS. https://github.com/utaal/verified-nrkernel. Accessed Feb. 2025. 81

[163] Verified splinter db. https://github.com/vmware-labs/verified-betrfs/tree/main/Splinter. Accessed Feb. 2025. 81

[164] Verified storage. https://github.com/microsoft/verified-storage. Accessed Feb. 2025. 81

[165] Andrei Voronkov. The anatomy of vampire: Implementing bottom-up procedures with code trees. *Journal of automated reasoning*, 1995. 8

[166] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*. Citeseer, 1990. 31

[167] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, 2023. 1

[168] Andrew S. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016. 65

[169] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*. Springer International Publishing, 2021. 9

[170] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *ACM Conference on Computer and Communications Security (CCS '17)*, 2017. 52

[171] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 221–239, 2020. 1

[172] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn Heule, and Bryan Parno. Context pruning for more robust smt-based program verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*, October 2024. 4, 122

[173] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, October 2023. 4, 48, 122

[174] Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. Galápagos: Developing Verified Low-Level Cryptography on Heterogeneous Hardware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2023. 1, 3, 41, 73

[175] Yi Zhou, Amar Shah, Zhengyao Lin, Marijn Heule, and Bryan Parno. Cazamariposas: Automated instability debugging in smt-based program verification. In *Conference on Automated Deduction*, 2025. 122

[176] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VERISMO: a verified security module for confidential VMs. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2024. 81

[177] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. 1, 19, 52

Appendices

A. Mutant Success Rates

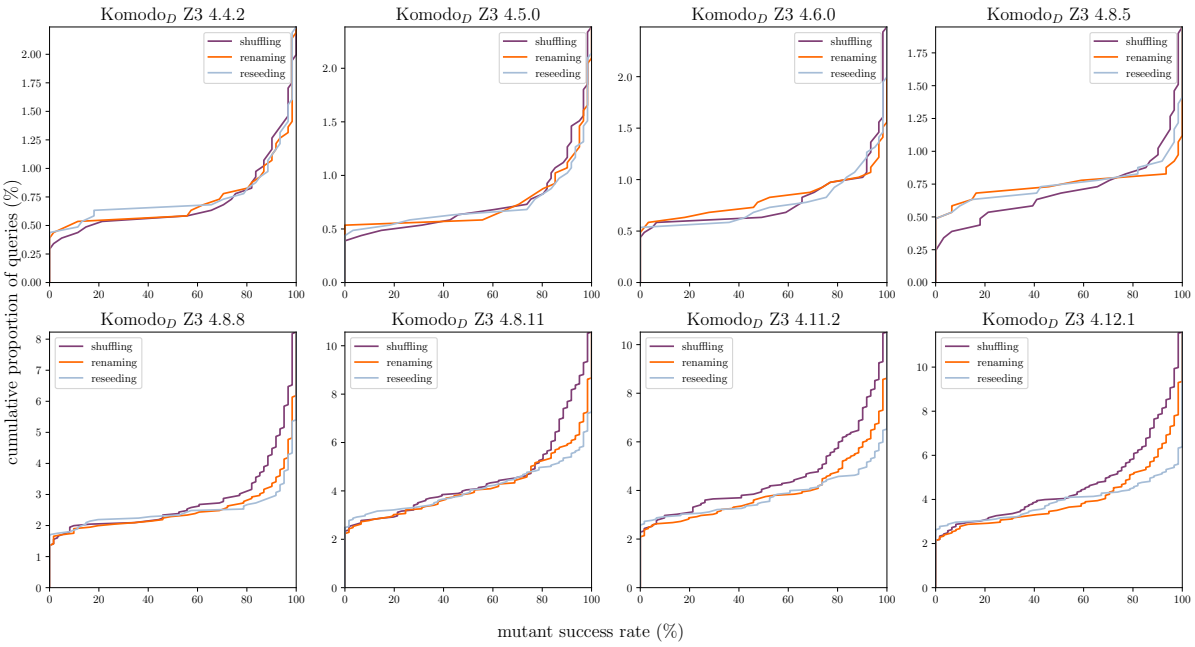


Figure 1: **Mutant Success Rates for Komodo$_D$**

Our stability categorization is based on the mutant success rate. (Recall the intuition behind the mutant success rate in Tab. 6.1.) Here we show the CDF of the mutant success rates in Komodo$_D$. Most of the queries have a mutant success rate of 100%, so the y-axis is adjusted to show the details of other cases. In later versions of Z3, the span of y-axis is larger, indicating more unstable or unsolvable queries. shuffling also spans a larger range on the y-axis than the other methods.

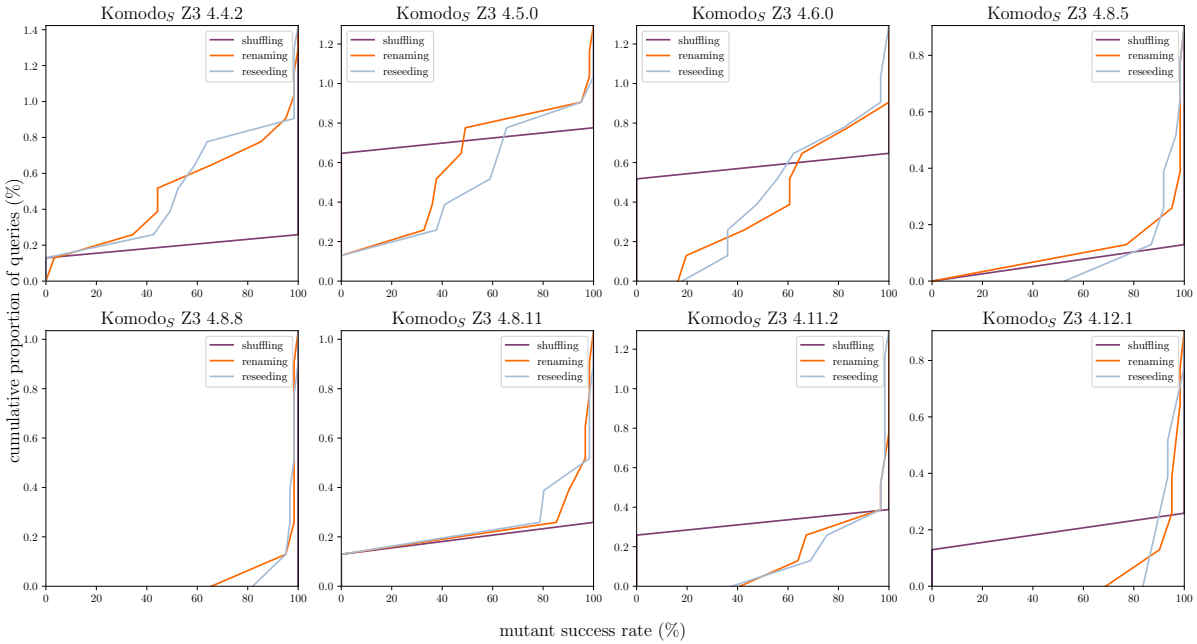


Figure 2: **Mutant Success Rates for Komodo$_S$**

The proportion of `unstable` queries is consistently low in Komodo$_S$, resulting in very few data points in the plots. Note **shuffling** is nearly a flat line, indicating that it almost finds no `unstable` queries. This is because **shuffling** only changes the order of consecutive `assert` commands in the query, and 99.5% of the Komodo$_S$ queries contains exactly one `assert`. This property is unique to Komodo$_S$.

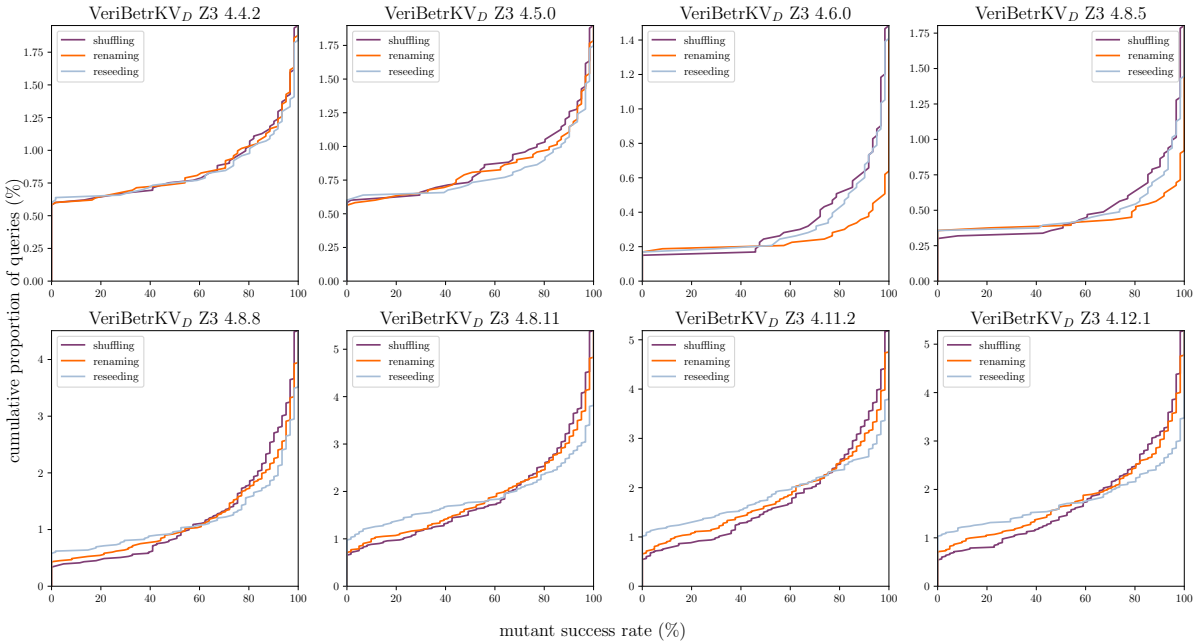


Figure 3: **Mutant Success Rates for VeriBetrKV$_D$.**

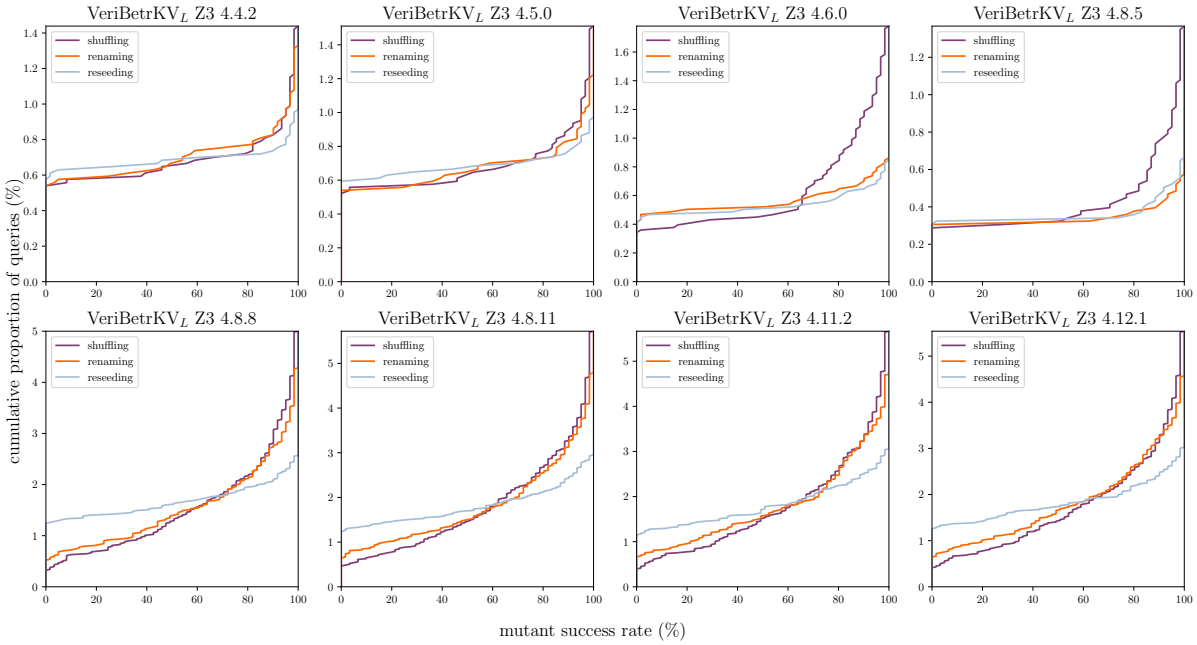


Figure 4: **Mutant Success Rates for VeriBetrKV$_L$.**

We observe similar patterns as in Fig. 1: in later versions of Z3, instability increases, and **shuffling** finds more unstable queries than other methods.

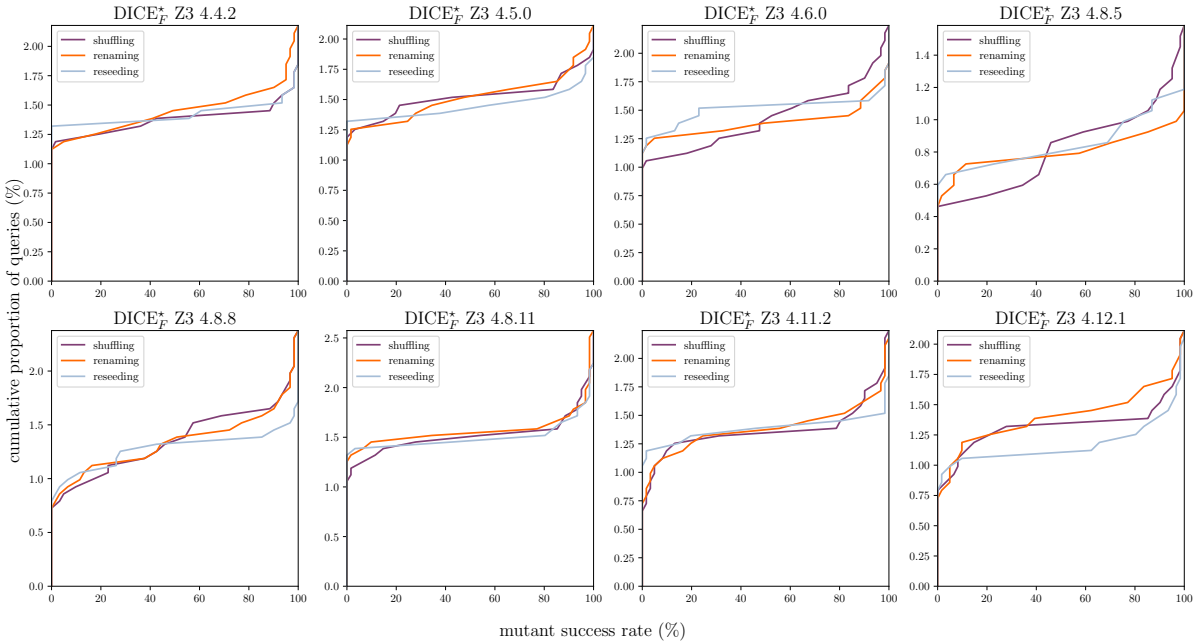


Figure 5: **Mutant Success Rates DICE$^{\star}_{F}$.**

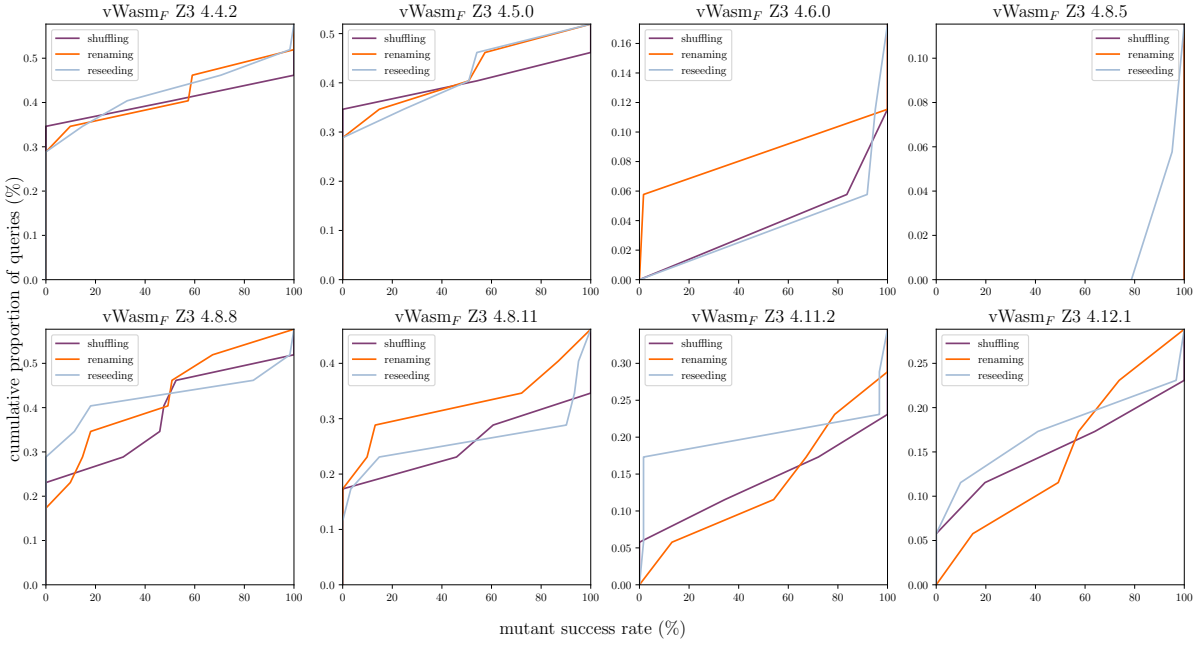


Figure 6: **Mutant Success Rates for vWasm$_F$.**

Instability exists in DICE$^{\star}_{F}$ but is not as severe as in Komodo$_D$, VeriBetrKV$_D$, and VeriBetrKV$_L$. The differences between mutation methods are not obvious. Meanwhile, vWasm$_F$ is overall stable, which results in a small number of data points in the plots.

B. Comparison on Time Limit Choices

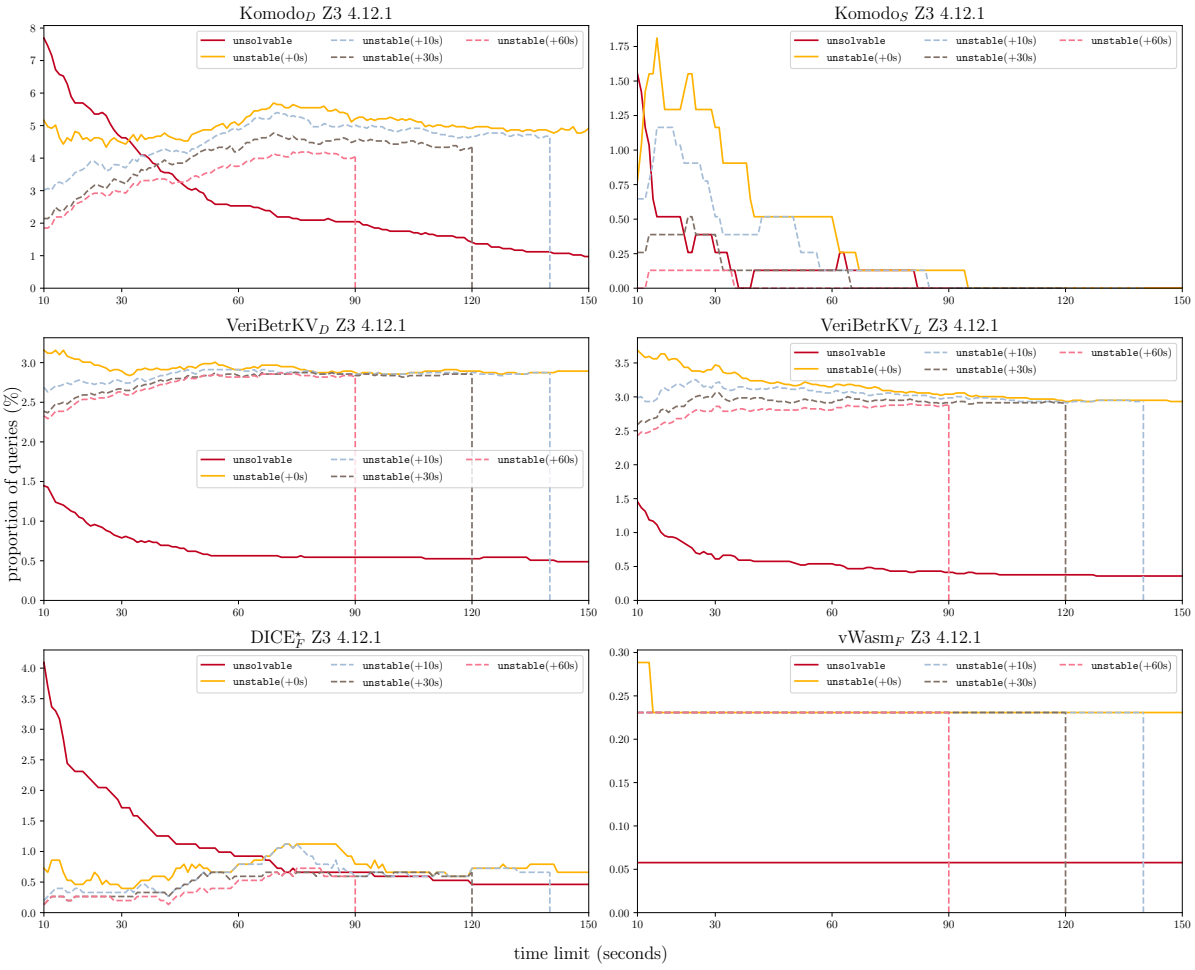


Figure 7: **Comparison on Time Limit Choices.**

We extend Fig. 6.3. Here we present the results from all six projects using the most recent version of Z3 and a 150s time limit. Note the step curves should all converge as T_{lim} approaches infinity, since the differences between adding 5s, 30s, and 60s become negligible. Such convergence is captured in all projects except Komodo$_D$, where 150s is insufficient. Note the curves often do not converge to 0%. For example, in VeriBetrKV$_L$, the proportion of `unstable` queries narrows to around 3% after 100s, but it does not further decrease. Therefore, increasing T_{lim} helps mitigate instability, but it has diminishing return beyond a certain threshold.

C. Comparison on Mutation Methods

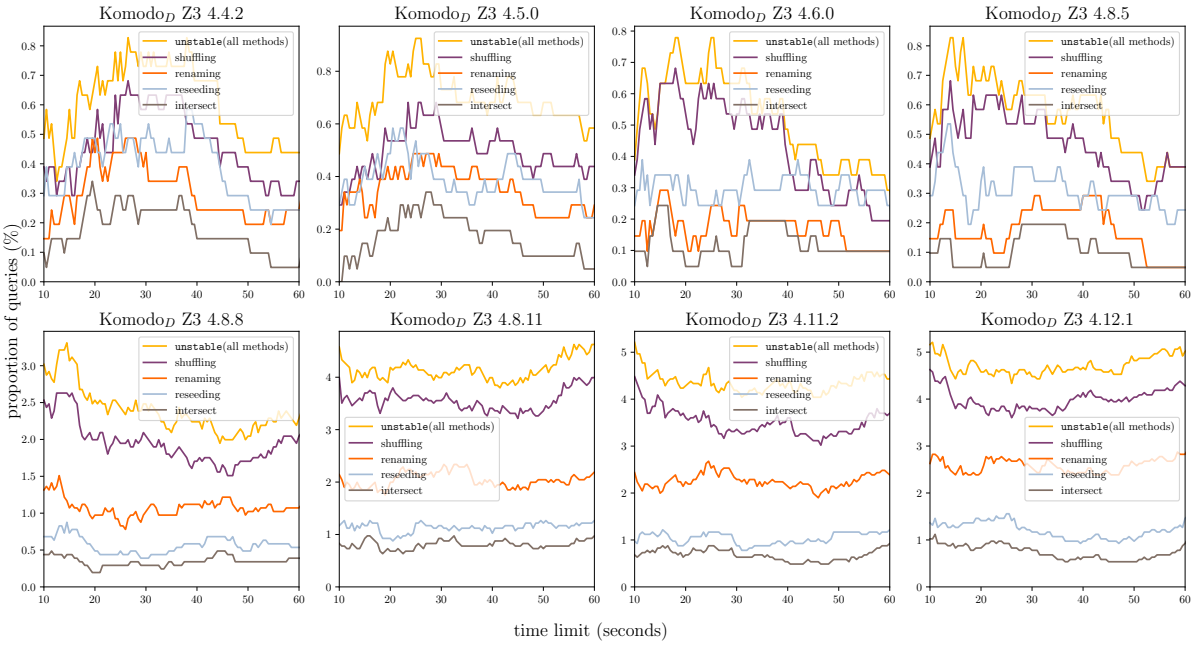


Figure 8: **Comparison on Mutation Methods for Komodo$_D$.**

We extend the results from Fig. 6.4 on Komodo$_D$ with all solvers versions. Recall that each plot shows the proportion of `unstable` queries from each mutation method. Note the scale of the y-axis is different for each plot, where older solver versions have fewer `unstable` queries. Differences between mutation methods also become pronounced as we move to newer solver versions, with **shuffling** being the most effective, and **reseeding** being the least effective.

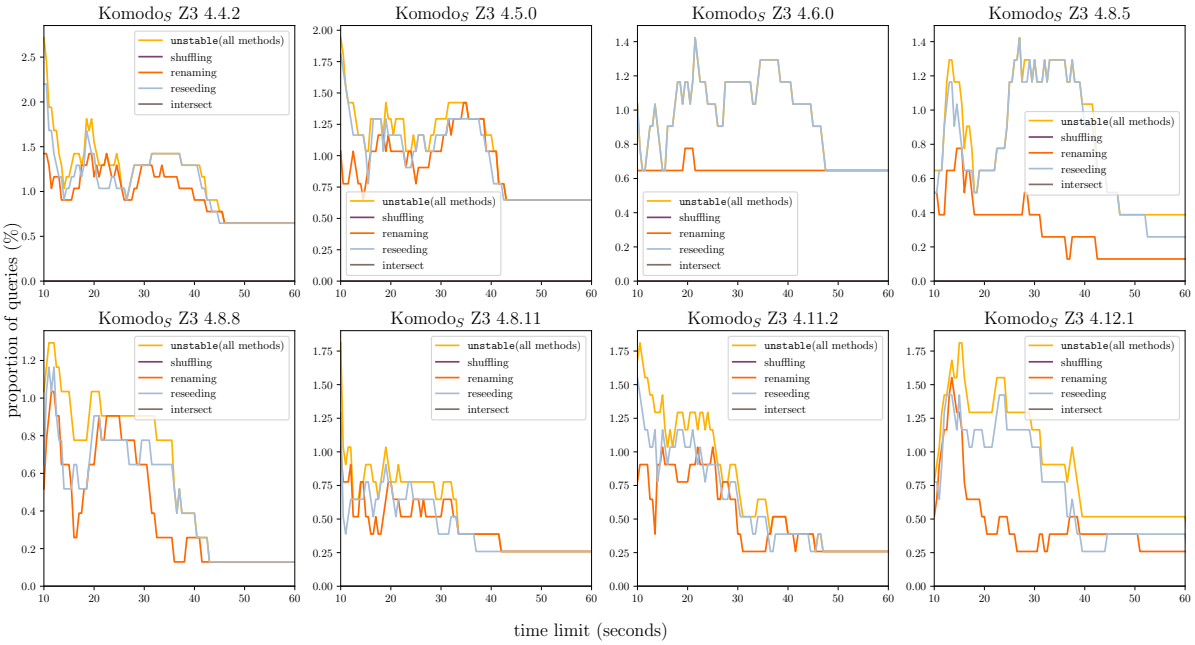


Figure 9: **Comparison on Mutation Methods for Komodo$_S$.**

In Komodo$_S$, the proportion of `unstable` queries is consistently low, and differences between mutation methods are not evident. We note that **shuffling** disappears in the plot, due to the reason explained in Fig. 2.

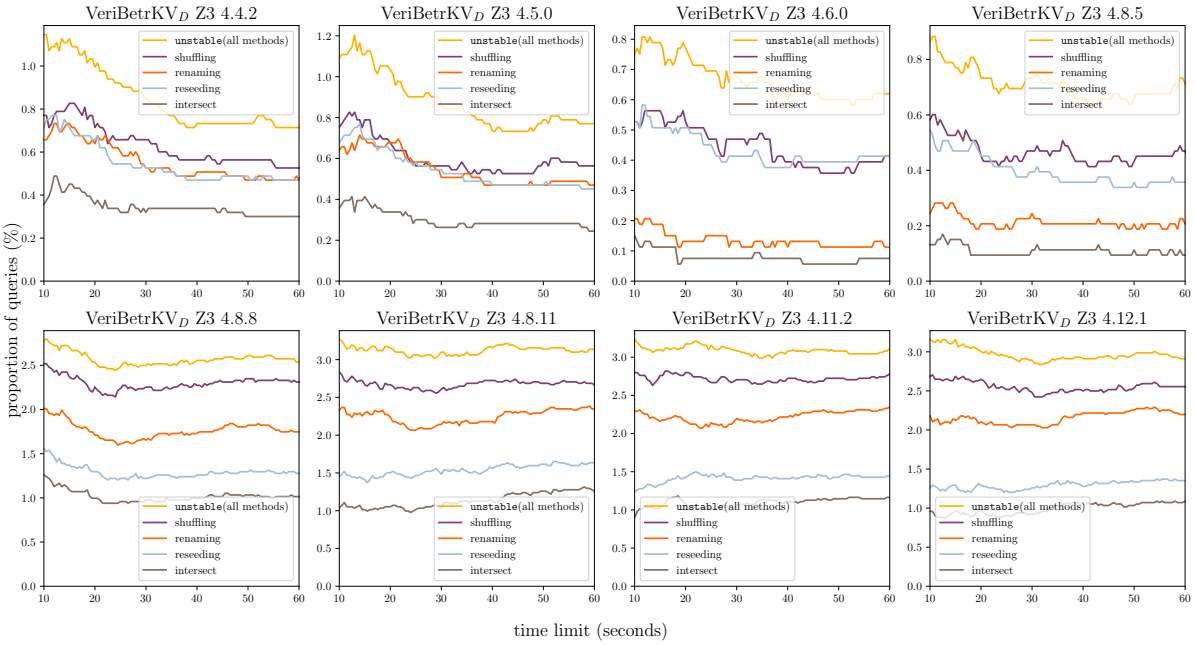


Figure 10: **Comparison on Mutation Methods for VeriBetrKV$_D$.**

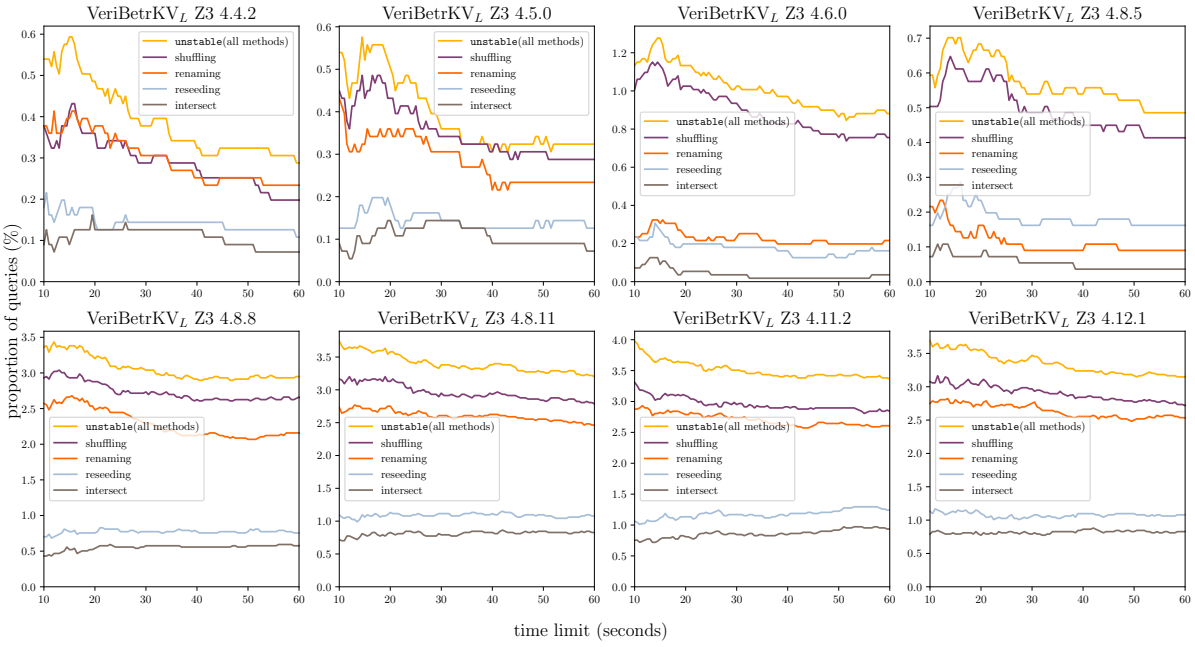


Figure 11: **Comparison on Mutation Methods for VeriBetrKV$_L$.**

In VeriBetrKV$_D$ and VeriBetrKV$_L$, we observe similar trends as in Komodo$_D$ from Fig. 8, where the overall instability and the differences between mutation methods increase as we move to newer solver versions.

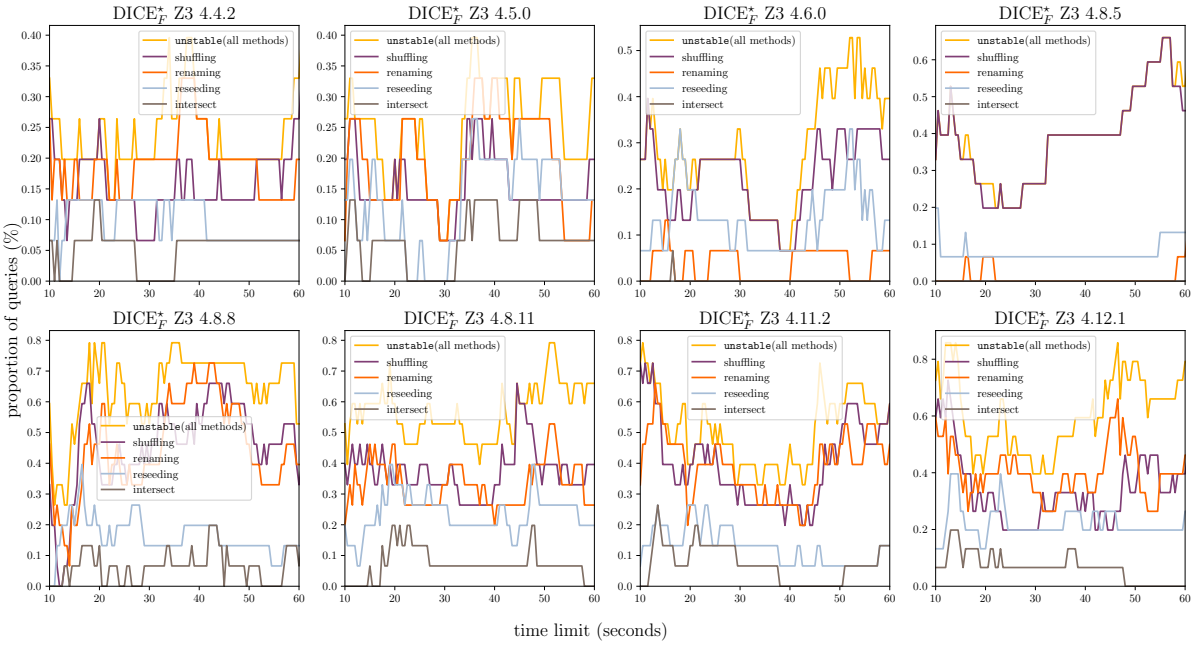


Figure 12: **Comparison on Mutation Methods for DICE$^\star_F$.**

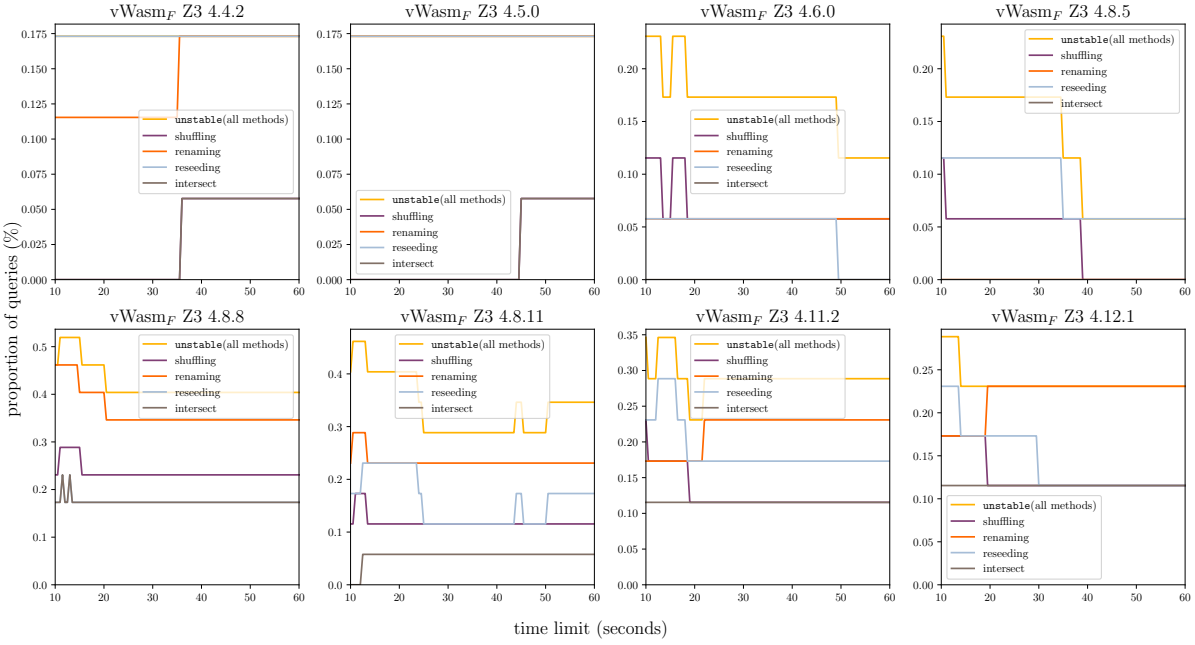


Figure 13: **Comparison on Mutation Methods for vWasm$_F$.**

The proportion of `unstable` queries is generally low for DICE$^\star_F$, but the proportion slightly increases as we move to newer solver versions. No obvious differences between mutation methods observed. The proportion of `unstable` queries is very low in all solver versions tested for vWasm$_F$.

D. Degree of Stability

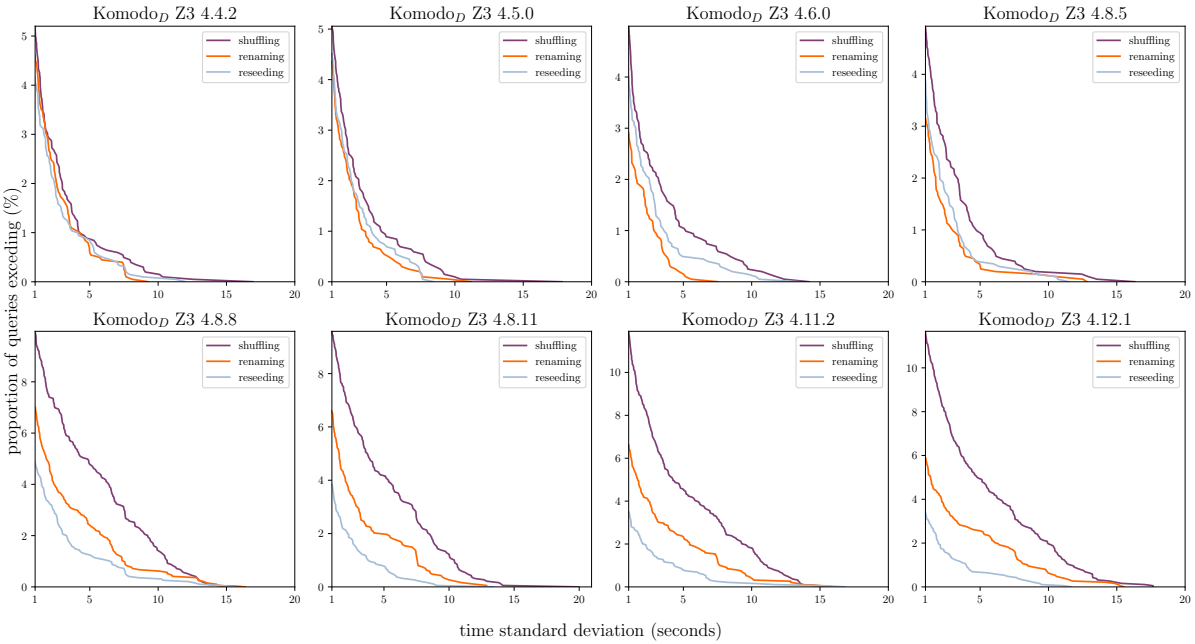


Figure 14: **Degree of Stability for Komodo$_D$.**

We show the results of Fig. 6.5 on Komodo$_D$ with all solver versions. Recall each plot shows the distribution of **Mutant Time Deviation** over the `stable` queries only. Note the scale of the y-axis is different for each plot, where older solver versions have fewer `stable` queries with large deviations. The differences between mutation methods are more pronounced in newer solver versions, with **shuffling** being the most effective, and **reseeding** being the least effective.

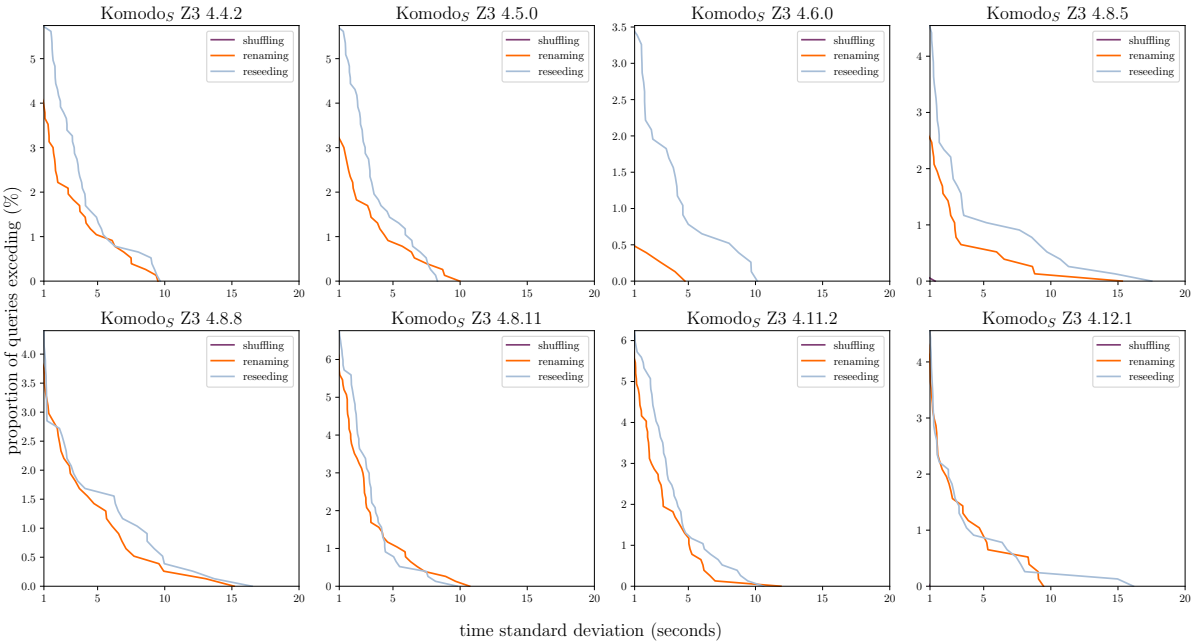


Figure 15: **Degree of Stability for Komodo$_S$.**

Komodo$_S$ is one of the more stable projects, but there are still non-negligible variations in mutant response time for `stable` queries. Note **shuffling** also disappears in the results as in Fig. 9.

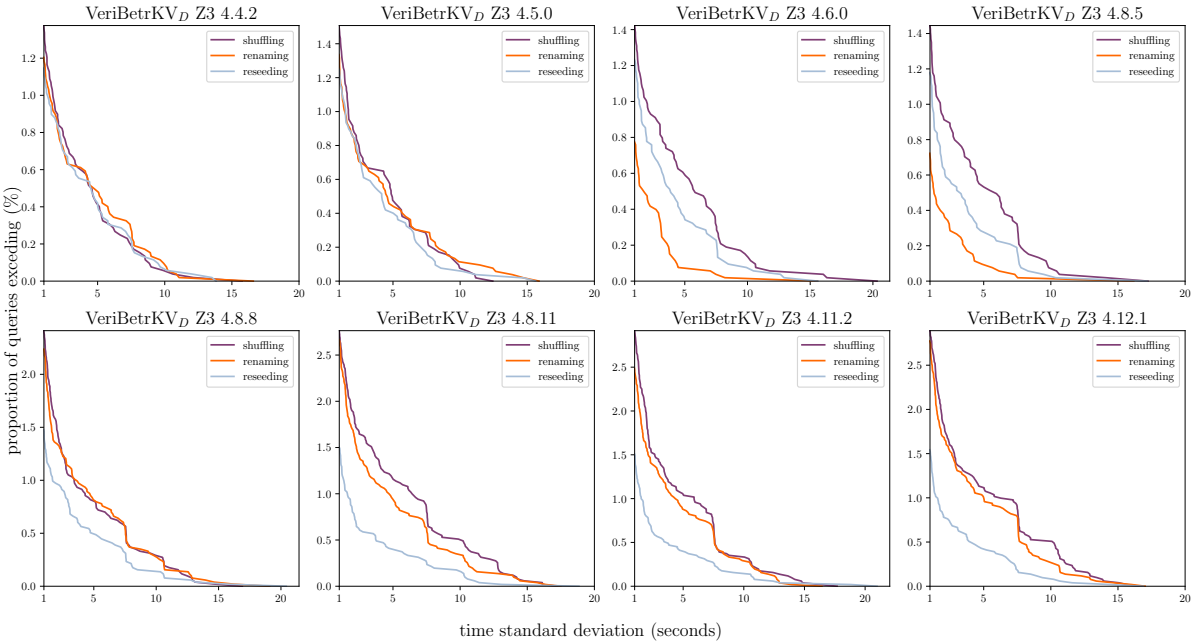


Figure 16: **Degree of Stability for VeriBetrKV$_D$.**

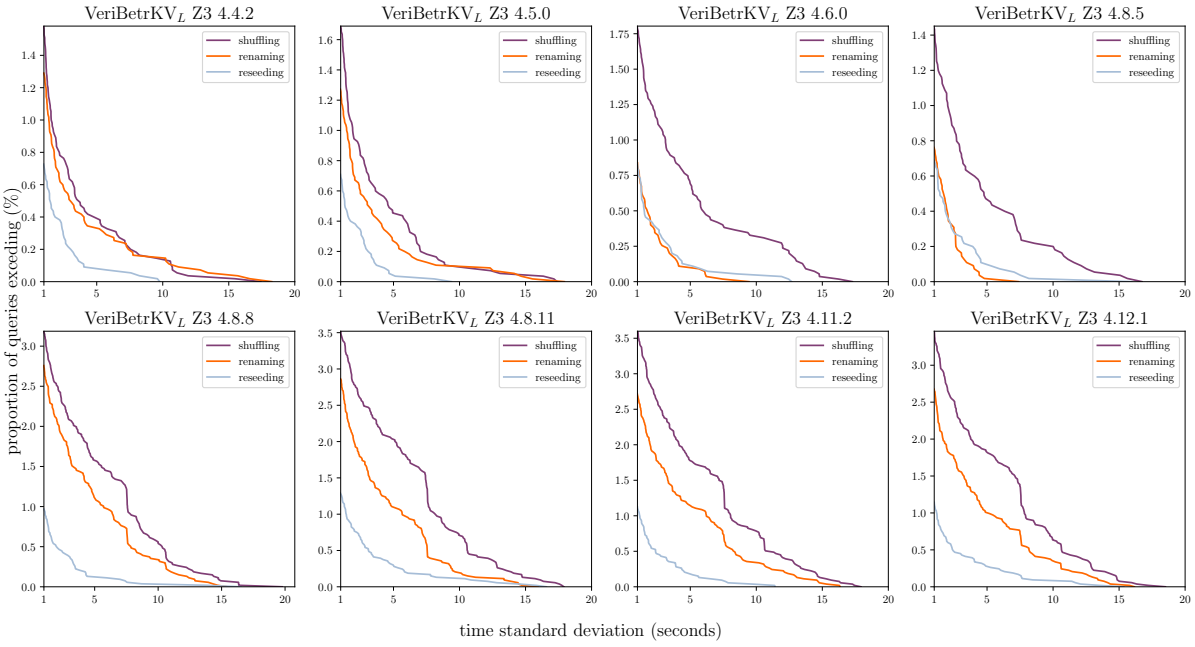


Figure 17: **Degree of Stability for VeriBetrKV$_L$.**

In VeriBetrKV$_D$ and VeriBetrKV$_L$, we observe similar trends as in Komodo$_D$ from Fig. 14 , where newer solver versions tend to have more `stable` queries with large deviations. Furthermore, the differences between mutation methods are also more pronounced, where **shuffling** and **renaming** have larger than **reseeding**.

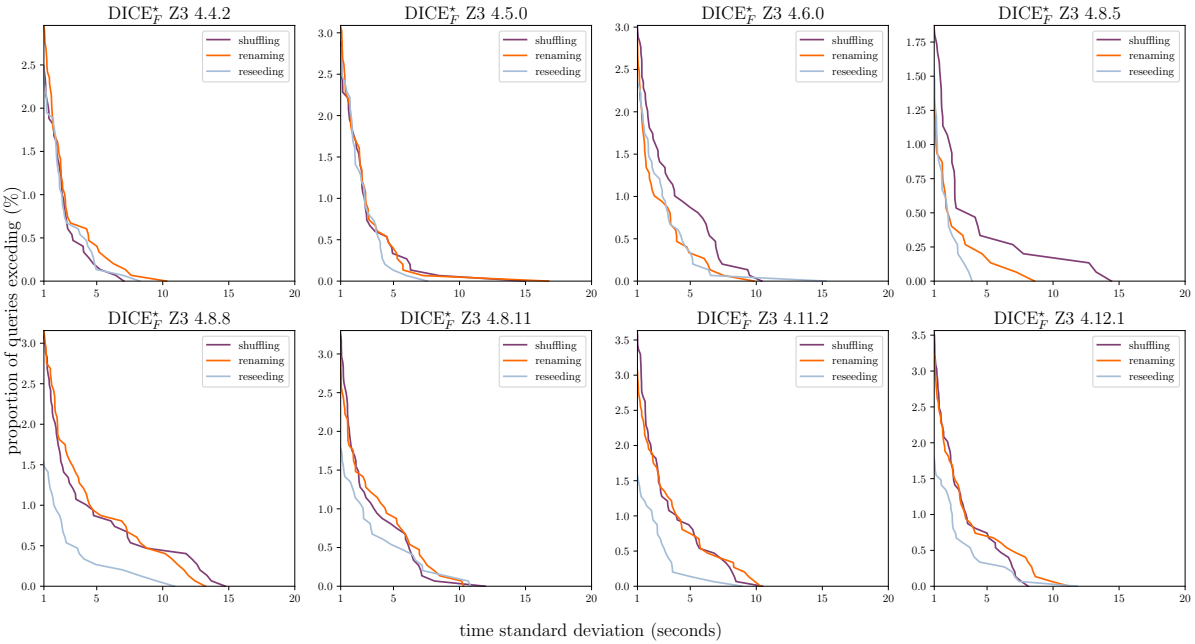


Figure 18: **Degree of Stability for DICE$^\star_F$.**

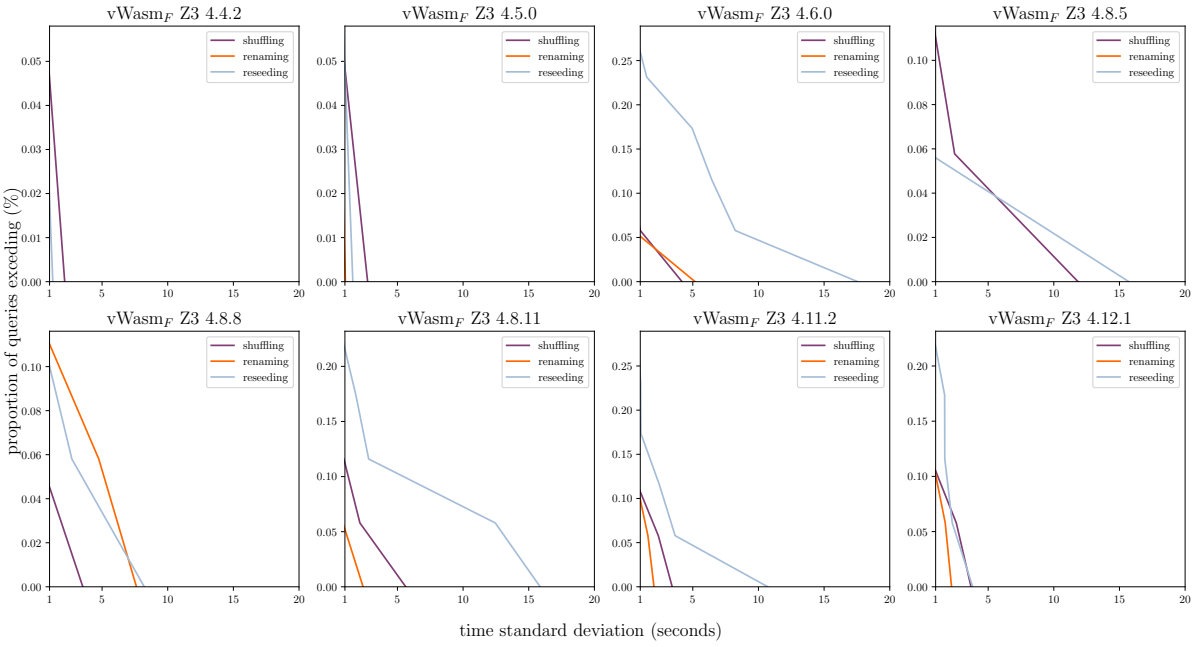


Figure 19: **Degree of Stability for vWasm$_F$.**

DICE$^\star_F$ is one of the more stable projects, but there are still variations in mutant response time for `stable` queries. vWasm$_F$ is the most stable project. Note the majority of the `stable` queries have standard deviations less than one second.

E. Comparison on Original and Mutant Queries

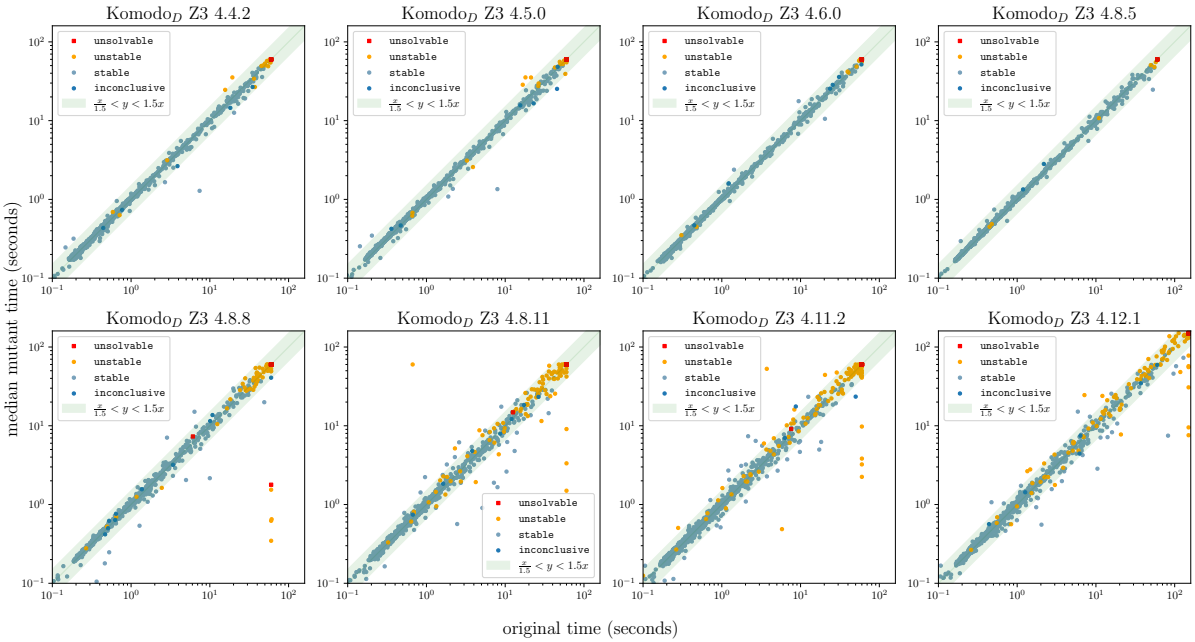


Figure 20: **Comparison on Original and Mutant Queries for Komodo$_D$.**

We show the results of Fig. 6.6 on Komodo$_D$ with all solver versions. Only the latest Z3 version has results from extended T_{lim} experiments. We observe some data points on the right side of each the plot, meaning there are some cases where the original query times out, but where median of the mutants does not. Note the log-log scale of the plots. Generally the performance of the original query and the median of its mutants are bounded within a 1.5 factor of each other.

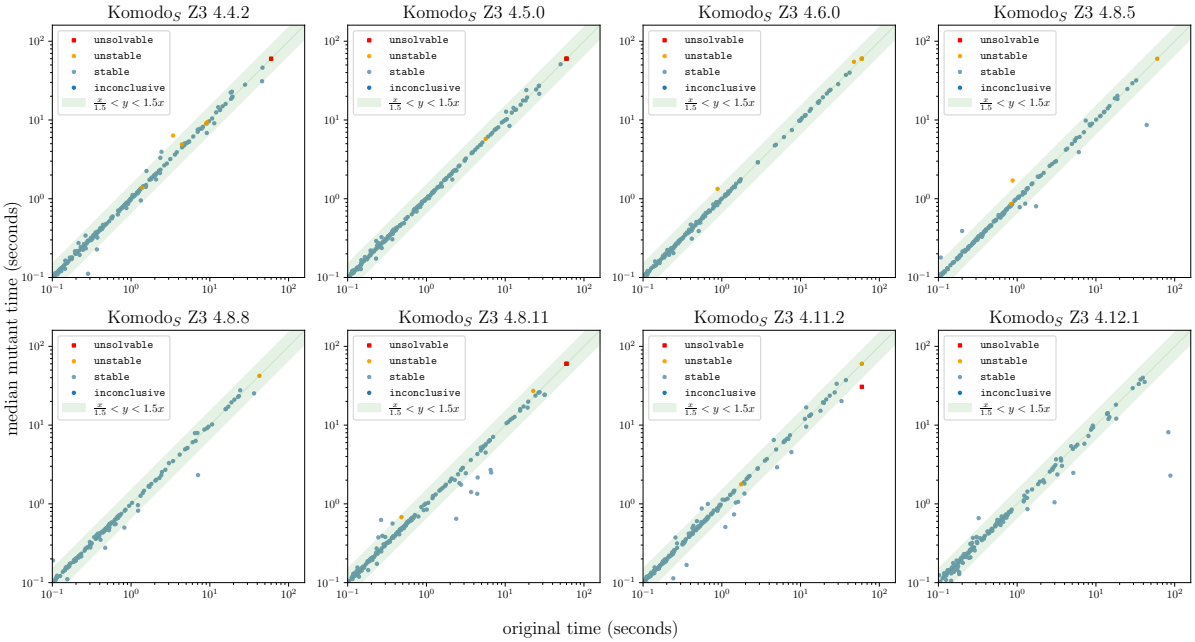


Figure 21: **Comparison on Original and Mutant Queries for Komodo$_S$.**

The performance of the original query and the median of its mutants are more tightly bounded within the highlighted region than in Fig. 20.

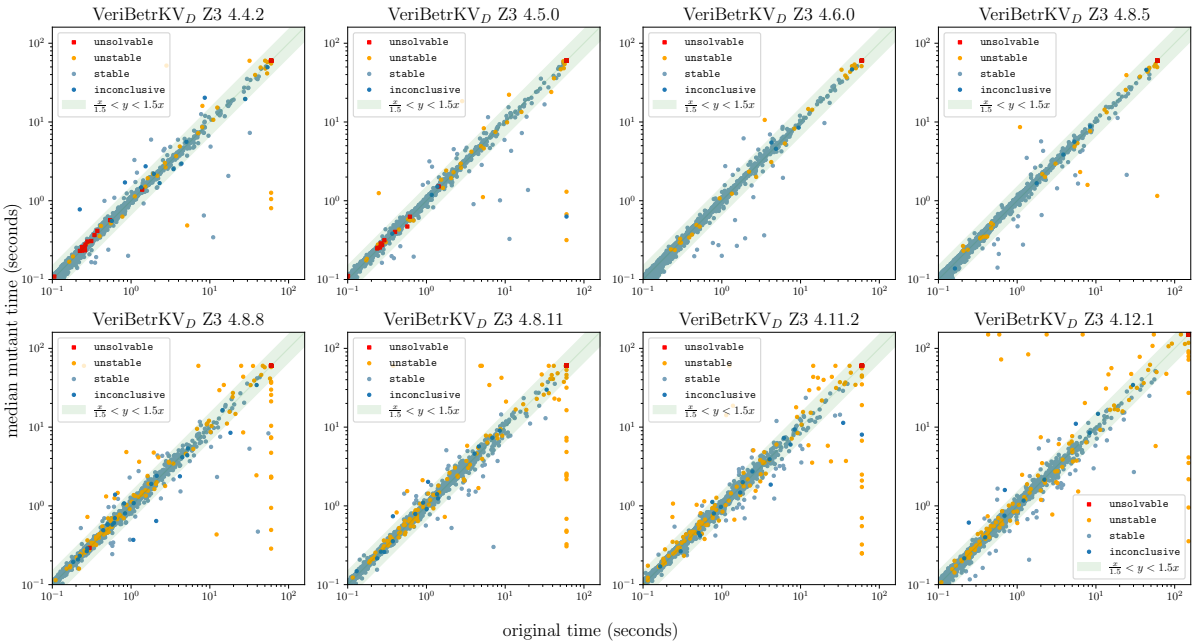


Figure 22: **Comparison on Original and Mutant Queries for VeriBetrKV$_D$.**

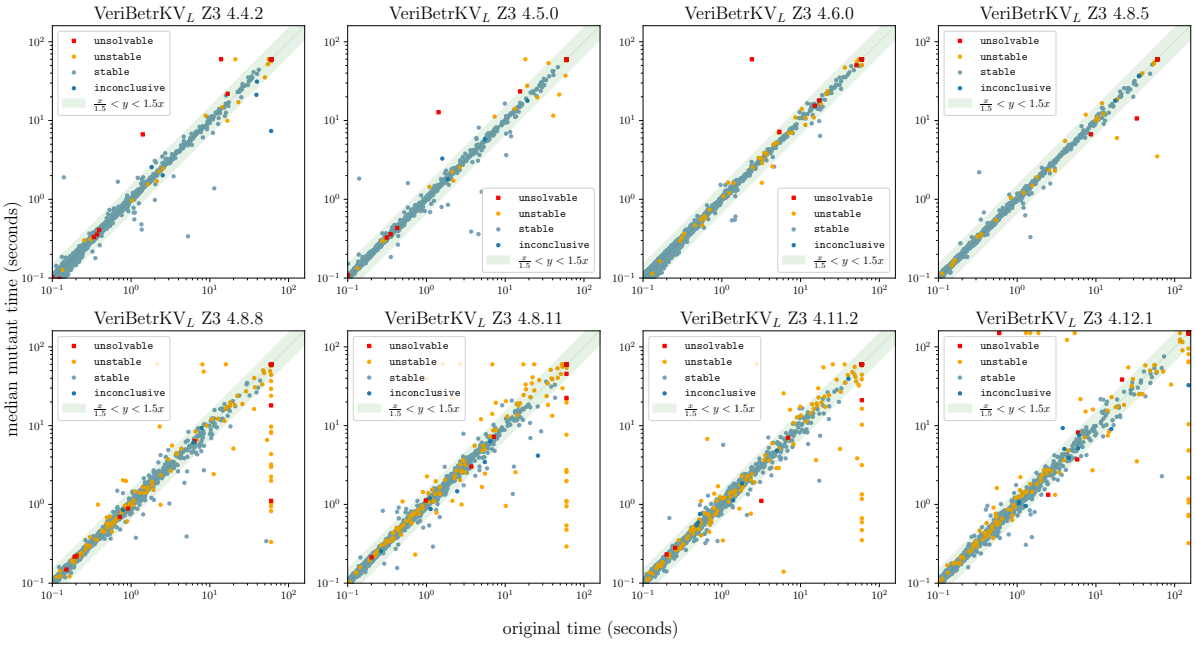


Figure 23: **Comparison on Original and Mutant Queries for VeriBetrKV$_L$.**

In VeriBetrKV$_D$ and VeriBetrKV$_L$, the data points are more scattered in the new solver versions. We also observe some data points on the top side of the plot, meaning there are some cases where the median of the mutants is worse than the original query. However, there is no obvious trend that the original query is consistently better or worse overall.

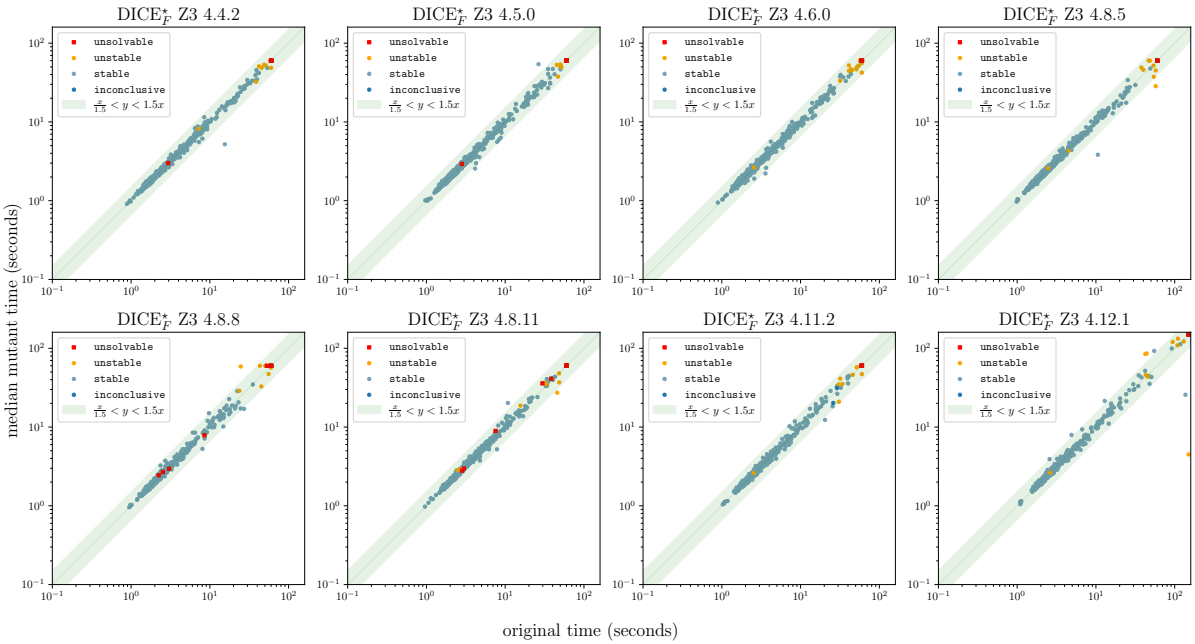


Figure 24: **Comparison on Original and Mutant Queries for DICE$^\star_F$.**

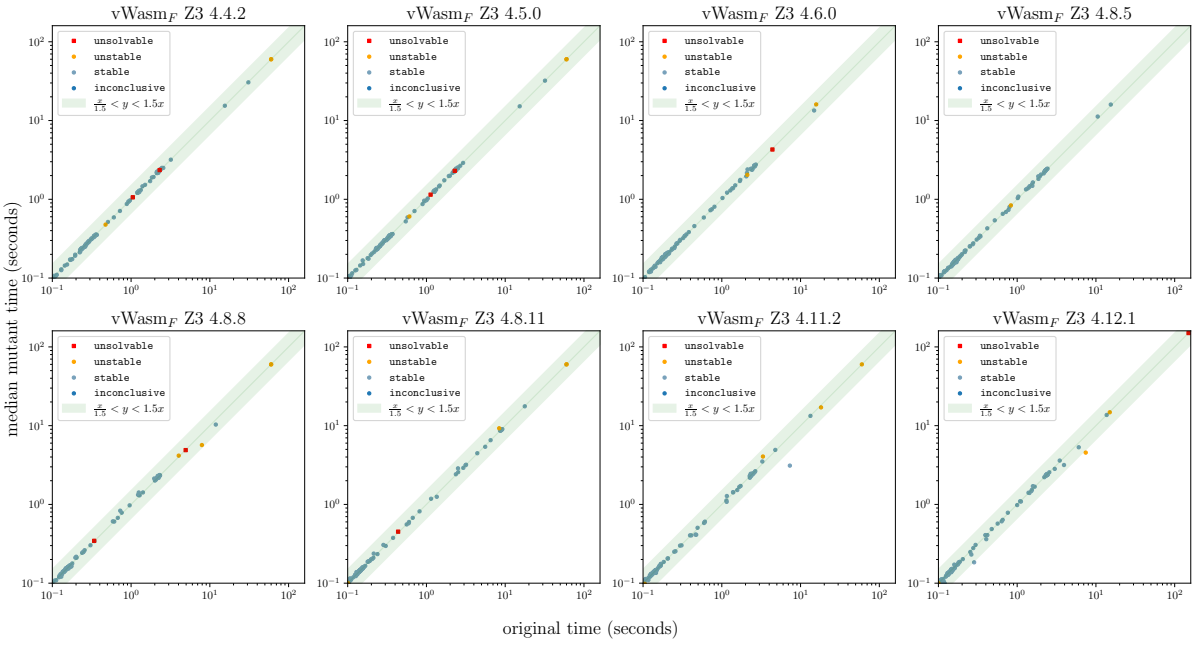


Figure 25: **Comparison on Original and Mutant Queries for vWasm$_F$.**

In DICE$^\star_F$, performance is generally bounded within the highlighted region. Meanwhile, almost no query finishes $\leq$ 1s, which could be due to the fact that most queries in DICE$^\star_F$ are larger than 10MB in size. vWasm$_F$ tends to produce faster running queries, and the performance differences between the original query and its mutants is almost negligible.