Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Supporting Alternative SMT Solvers in Viper

Practical Work

Lasse F. Wolff Anthony

April 28, 2022

Advisor: Dr. Malte Schwerhoff
Department of Computer Science, ETH Zürich

Abstract

Program verification deals with the problem of proving or disproving the correctness of a program with respect to its specifications. Such verification can be automated thanks to verification infrastructures like Viper. Viper is a tool chain and infrastructure for program verification that natively supports permission-based reasoning in its intermediate language. It also consists of the automated verification backend called Silicon that is based on symbolic-execution. Silicon ultimately relies on an SMT solver to discharge any resulting proof obligations.

In this work, we introduce a new version of Silicon that only produces encodings that are fully conformant with the newest SMT-LIB standard. We add backend support in Silicon for multiple SMT solvers and introduce the cvc5 solver as an alternative to the default Z3 solver. We test and benchmark our changes to Silicon to show that SMT-LIB conformance does not hinder performance compared to the original version. Finally, we show that cvc5 can fix several open issues in Silicon and may prove to be a useful alternative to the Z3 solver for some inputs.

Contents

Contents iii

1 Introduction 1

2 Viper 3
2.1 Silicon 4

3 Satisfiability Modulo Theories Solvers 7
3.1 SMT-LIB Standard: Version 2.6 8
3.2 Z3 9
3.3 cvc5 9

4 Implementation 11
4.1 SMT-LIBv2.6 Conformance 11
4.2 Backend Support for Multiple SMT Solvers 12
4.2.1 Supporting cvc5 13

5 Evaluation 15
5.1 Completeness and Soundness Testing 16
5.2 Benchmarking 18
5.2.1 Original Against SMT-LIBv2.6 Flavored Silicon 18
5.2.2 Z3- Against cvc5-Based Silicon 18
5.2.3 To Force or Not To Force State Saturation Checks . . . 20

6 Discussion 23
6.1 Unsoundness With cvc5 23
6.2 Completeness and Improper Test Annotations 24

7 Conclusion 27
7.1 Future Work 27

A Per Test Directory Benchmarks And Statistics **29**
A.1 Statistics Per Silicon Flavor 29
A.2 Original Against SMTLIBv2.6 Flavored Silicon 29
A.3 Z3- Against cvc5-Based Silicon 30
A.4 To Force or Not To Force State Saturation Checks 31

Bibliography **49**

Chapter 1

Introduction

Program verification is a critical component of ensuring that programs behave as intended, as per their correctness specifications. Automating these verification techniques require scalable infrastructures on which such automated program verification tools can be built. Viper [1] [13], or Verification Infrastructure for Permission-based Reasoning, is one such infrastructure. More precisely, Viper is a tool chain and infrastructure for program verification that is being developed by the Programming Methodology Group at ETH Zurich, Switzerland [2]. It consists of the Viper intermediate language, based on separation logic to encode verification problems, automated verifiers for the language, and several example front-end tools for popular languages such as Python, Rust, and Go.

Viper includes two automated verification backends: (1) Silicon [3] [16], a verifier based on symbolic execution, and (2) Carbon [4], a verifier based on verification-condition generation. Both of these verifiers ultimately rely on an SMT solver to discharge resulting proof obligations. Silicon interacts directly with an SMT solver, which prior to this work was the Z3 Theorem Prover [5] [12]. Carbon instead uses the intermediate verification language Boogie [6] [3] to abstract over the underlying SMT solver, which again in Boogie is Z3.

Most widely-used SMT solvers offer support for the SMT-LIB command language (or a dialect hereof) as input as defined by the SMT-LIB standard [6], but use different combinations of logic-specific sub-solvers, strategies, and heuristics to solve different problems, which in turn affects their com-

[1] http://viper.ethz.ch

[2] https://www.pm.inf.ethz.ch/

[3] https://github.com/viperproject/silicon

[4] https://github.com/viperproject/carbon

[5] https://github.com/Z3Prover/z3

[6] https://github.com/boogie-org/boogie

pleteness and efficiency. Hence, the completeness and efficiency of Silicon, Carbon, and, by extension, the automated program verification capabilities offered by Viper are closely tied to that of the underlying SMT solver. Introducing support for multiple SMT solvers in Viper's verification backends may result in improved performance for certain classes of inputs.

In this work, we focus on the symbolic-execution-based verification backend Silicon. We introduce a new version of Silicon that produces only SMT commands that conform to the newest SMT-LIB standard. We add extended backend support for multiple SMT solvers and include the recently released cvc5 solver [7] [2] as an alternative to Z3. Finally, we present several tests and benchmarks to show that this new version of Silicon is equally performant as the previous version using Z3-specific SMT commands and that cvc5 can be used as an alternative solver to discharge resulting proof obligations in Silicon.

The structure of this document is as follows: In Chapter 2, we begin with a brief introduction to Viper and the symbolic-execution-based verification backend Silicon. Next, in Chapter 3, we provide some background information on SMT solvers and the SMT-LIB standard, contextualizing the required changes made in Silicon. In Chapter 4, we report the implementation details of modifying Silicon to conform to the newest SMT-LIB standard as well as how the Silicon codebase can be generalized to support multiple SMT solvers including cvc5. Chapter 5 describes the extensive testing and benchmarking performed to evaluate our implementations of Silicon. The results of the evaluation are then further discussed in Chapter 6. Ultimately, Chapter 7 draws a conclusion on the contributions made and outlines directions for future work.

[7] https://cvc5.github.io/

Chapter 2

Viper

The Viper verification infrastructure is visualized in Figure 2.1 together with the contributions to the Silicon-based pipeline of the stack found in this work. Viper has already been successfully used to build several front-end verification tools, including verifiers such as the Nagini verifier for Python [1] [10], the Prusti verifier for Rust [2] [1], and the Gobra verifier for Go [3] [18].

The Viper intermediate language is a powerful but simple imperative programming language with extensive support for permission-based reasoning. It natively supports a notion of a program heap and access permissions to heap locations. The language allows for convenient expression of verification problems and techniques used in automated program verification. In Listing 2.1 we show a simple example of a Viper program without access permissions. The program proves (after verification with either Silicon or Carbon) that our iterative implementation of computing the nth Fibonacci number is indeed correct. We see that preconditions are specified using the `requires` keyword, postconditions with `ensures`, and loop invariants required to prove our postcondition can be specified using `invariant`. Readers may consult the official Viper tutorial [4] for an extensive introduction to the Viper intermediate language and its many features, including permission-based reasoning.

As mentioned previously, Viper includes two automated verification backends. One is the Carbon verifier based on verification-condition generation, where the program under consideration is translated into a set of logical formulas whose validity implies that the program must satisfy the specified correctness properties. Carbon uses Boogie and the Boogie intermediate verification language [11] for this purpose. The other verifier is the

[1] https://www.pm.inf.ethz.ch/research/nagini.html
[2] https://www.pm.inf.ethz.ch/research/prusti.html
[3] https://www.pm.inf.ethz.ch/research/gobra.html
[4] http://viper.ethz.ch/tutorial/

```
0  // Fibonacci recurrence relation
1  function fib(n: Int): Int
2  {
3    (n < 2) ? n : fib(n-2) + fib(n-1)
4  }
5
6  // Iteratively computes nth Fibonacci number
7  method iter_fib(n: Int) returns (res:Int)
8    requires n >= 0
9    ensures res == fib(n) // prove equivalent to recursive
       definition
10 {
11   res := 0
12   var i: Int := 0
13   var next_fib: Int := 1
14
15   while(i != n)
16     invariant i >= 0
17     invariant res == fib(i)
18     invariant next_fib == fib(i+1)
19   {
20       var old_res: Int := res
21       res := next_fib
22       next_fib := old_res + res
23       i := i + 1
24   }
25 }
```

Listing 2.1: Simple Viper program that after verification proves that our implementation of an iterative method for computing the nth Fibonacci number is correct.

symbolic-execution-based Silicon that explores different program execution paths. Silicon is the focus of this work, and we briefly introduce the verifier in the following section.

2.1 Silicon

Silicon is an automated verification backend for Viper based on sound symbolic execution. The verifier directly translates Viper programs into (usually short) SMT commands in a stepwise fashion and then (frequently) issues these commands to an underlying SMT solver to discharge proof obligations that arise during verification. The interaction between Silicon and the underlying SMT solver is based on standard I/O streams, where Silicon writes to and reads from files in (some dialect) of the SMT-LIB command language. Prior to this work, Silicon was only compatible with Z3 as the underlying SMT solver due to utilizing Z3-specific SMT commands that were not defined in the official SMT-LIB command language and were therefore not supported by other SMT solvers. Interested readers may refer to Schwerhoff

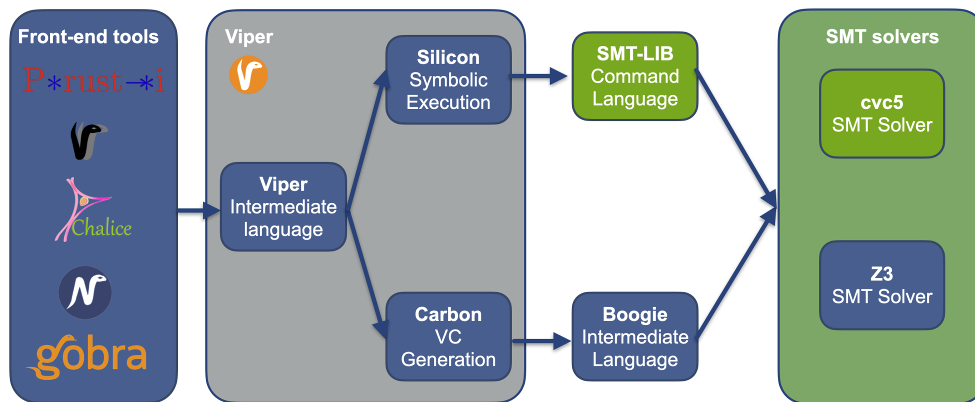

Figure 2.1: The Viper verification infrastructure. Front-ends encode into the Viper intermediate language which can then be verified using one of two verification backends, either Silicon based on symbolic execution or Carbon based on verification condition generation. Both of these automated verifiers rely on other tools; ultimately they both discharge proof obligations using the Z3 SMT solver (prior to this work). The contributions of this work (shown in green) are to the Silicon pipeline of this stack, where we generalize Silicon to produce encodings only in the SMT-LIB command language and add backend support for multiple SMT solvers as well as introduce an alternative solver in the form of cvc5.

[16] for a thorough description of Silicon's inner workings and design.

Chapter 3

Satisfiability Modulo Theories Solvers

Satisfiability Modulo Theories (SMT) is an area of automated deduction that deals with the problem of deciding the satisfiability of first-order formulas within (modulo) some logical background theory [4]. SMT solvers are tools that can be used to help with this decision problem. These SMT solvers often utilize the lazy approach that combine propositional satisfiability (SAT) solvers with theory-specific solvers. Unfortunately, the SMT problem is generally NP-hard, and for some of these logical theories, undecidable. Therefore, depending on the strategies and heuristics used, SMT solvers have varying levels of completeness and efficiency across different tasks and combinations of background theories. Nonetheless, SMT solvers have become an invaluable tool in a variety of applications including program verification with Viper.

Some publicly available SMT solvers that are under active development, at time of writing, are: Alt-Ergo [1] [8], Boolector [2] [14], cvc5, veriT [3] [7], Yices 2 [4] [9], and Z3.

In the following sections, we will introduce SMT-LIB, the SMT-LIB Standard in its current version 2.6, and the two SMT solvers, Z3 and cvc5, that are suitable for use in Silicon. Readers unfamiliar with SMT may refer to Barrett and Tinelli [4] for background knowledge, since facilitating a deeper understanding of SMT is outside the scope of this work.

[1] https://alt-ergo.ocamlpro.com/
[2] https://boolector.github.io/
[3] https://verit.loria.fr/
[4] https://yices.csl.sri.com/

```
0  > (set-option :print-success true)
1  success
2  > (set-logic QF_LIA) ; integer arithmetic
3  success
4  > (declare-const a Int) ; declare integer constant a
5  success
6  > (declare-const b Int) ; declare integer constant b
7  success
8  > (assert (> (- a b) (+ a (- b) 2))) ; assert a - b > a + (-b)
       + 2
9  success
10 > (check-sat)
11 unsat
```

Listing 3.1: Example interaction of using the SMT-LIB command language with an SMT solver. The symbol > indicates input to the solver.

3.1 SMT-LIB Standard: Version 2.6

The Satisfiability Modulo Theories Library or SMT-LIB [5] is an initiative that aims to facilitate research and development in SMT. The initiative provides: (1) standards for relevant logical background theories and a language with which to specify them, (2) an underlying logical and expression language, (3) a language for sub-logics, and (4) a command language for interfacing with SMT solvers, named the SMT-LIB command language. They also arrange an annual SMT solver competition SMT-COMP [5], where SMT solvers compete on a large collection of SMT-LIB benchmarks. The most recent version of the SMT-LIB Standard is version 2.6 (SMT-LIBv2.6) [6] which we will use in this work.

The SMT-LIB command language is built on S-expressions, a way of representing nested lists of tree-like data that is designed to be easy to parse and process. Readers familiar with the programming language Lisp or Common Lisp [17] should recognize the parenthesized syntax (parsers supporting Common Lisp should also work for SMT-LIB scripts). In SMT-LIB an S-expression is either a non-parenthesized token (e.g. `abc`) or a sequence of 0 or more S-expressions enclosed in parentheses (e.g. `(abc (123))`). The command language both defines a standard set of these S-expressions that the user may issue and a set of responses for SMT solvers such as: `success` for successful execution of a supported command, `unsupported` if the command is not supported by the solver, and `sat` and `unsat` for satisfiable and unsatisfiable formulas, respectively. We show an example in Listing 3.1. Readers may refer to the official standard [6] for further information on SMT-LIBv2.6 and the SMT-LIB command language.

[5] https://smt-comp.github.io

Table 3.1: Extract of Z3 commands and their equivalent in the SMT-LIB command language.

Explanation	Z3 command	SMT-LIB command	Remark
Logical implication	`implies`	`=>`	
Logical biconditional	`iff`	No equivalent	i
Nullary sort	`(declare-sort` s`)`	`(declare-sort` s `0)`	ii
Empty annotation	`(!` t`)`	Not allowed	
Pair-wise distinct	`(distinct ···)`	`(distinct` x_1 x_2 `···)`	iii

[i] Can e.g. be encoded as XNOR or a conjunction of two implications. [ii] Z3 can infer arities in some scenarios while SMT-LIBv2.6 requires all associated arities to be specified. Z3 also allows a similar shorthand for introducing algebraic datatypes. [iii] SMT-LIBv2.6 requires at least two arguments while Z3 allows any number.

3.2 Z3

The Z3 theorem prover, also referred to as Z3, is a high-performance SMT-LIBv2.6 conformant SMT solver developed at Microsoft Research. It is one of the most popular SMT solvers, being widely used in both research and industry, and has been in development since 2006. Z3 supports a wide array of logical background theories, and with its broad support for quantifiers, it is a common choice for program verification tools such as Boogie and Viper.

Z3 supports a superset of the SMT-LIB command language and introduces several convenient shorthand commands. Z3 also generally allows a much more lenient syntax than that specified in SMT-LIBv2.6. In Table 3.1 we highlight a few differences between the Z3 dialect and SMT-LIBv2.6 that are relevant for our work. We emphasize that Z3 is SMT-LIBv2.6 conformant and therefore supports both the SMT-LIB command language and the Z3-specific dialect.

3.3 cvc5

The cvc5 solver is the latest SMT solver in the cooperating validity checker (CVC) tools developed jointly by Stanford University and the University of Iowa. The solver builds on its successful predecessor CVC4 [6] and supports all standard SMT-LIB theories and many non-standard theories, including reasoning about quantified formulas. It is further fully SMT-LIBv2.6 conformant and therefore has support for the newest SMT-LIB command language.

Barbosa et al. [2] promisingly show that cvc5 frequently solves significantly more non-incremental SMT-LIB benchmarks than Z3 when comparing across the same divisions used for SMT-COMP 2021. We, however, note that Sili-

[6] Note the change in naming convention. Earlier solvers in the CVC family used capital letters, whereas cvc5 use lower-case letter to acknowledge the tool's advancement and the authors finding it "more visually appealing" [2, p. 2].

con relies on incremental solving and the benchmarks performed by Barbosa et al. therefore may not be representative of cvc5's performance in Silicon.

Chapter 4

Implementation

In this section, we cover the implementation details of modifying Silicon to only produce commands following the SMT-LIB command language, and generalizing the Silicon backend to support multiple SMT solvers including the cvc5 solver. We denote this new version of Silicon as SMT-LIBv2.6 Silicon. The interested reader is referred to pull request #609 in Silicon [1] for full details of the changes introduced to the Silicon codebase.

4.1 SMT-LIBv2.6 Conformance

As discussed in Section 2.1, Silicon interacts with SMT solvers through standard I/O by writing to and reading from files in the SMT-LIB command language. Internally, Silicon converts its own terms to S-expressions in the SMT-LIB command language using the `TermToSMTLib2Converter` class. In this class we e.g. define how to translate logical connectives such as logical conjunction or declaration of new sort symbols from its internal Silicon representation to S-expressions understood by the SMT solver.

The previous version of Silicon (prior to this work) translated its internal terms into a superset of the SMT-LIB command language that was supported by the Z3 solver (see Section 3.2). These Z3-specific commands, as mentioned earlier, introduce shorthand expressions such as `iff` for the logical biconditional and generally have a more lenient syntax than that specified by SMT-LIBv2.6. This results in commands that are not accepted by the majority of other SMT solvers. For that reason, we modify the Silicon backend to only produce commands in the SMT-LIB command language. Below, we highlight some changes that were needed primarily in the `TermToSMTLib2Converter` class, and some in the static config preambles forwarded to the SMT solver, to ensure this:

[1] https://github.com/viperproject/silicon/pull/609

1. Logical implication is encoded as `=>` instead of the Z3-specific keyword `implies`.

2. Logical biconditional is changed from the Z3-specific `iff` encoding to the logical conjunction of a material implication and a converse implication, i.e., we now encode $(P \iff Q)$ as $((P \implies Q) \wedge (Q \implies P))$, since SMT-LIBv2.6 does not define the biconditional. This encoding is also equivalent to the XNOR boolean operator; both showed similar performance in initial testing.

3. Explicit arities are introduced. Z3 can often infer arity by itself but SMT-LIBv2.6 requires associated arities to be specified. E.g., when introducing a new nullary sort symbol s, we now encode it as `(declare-sort` s `0)` instead of `(declare-sort` s`)`.

4. Annotations, defined using `(!` $t\ \alpha_1\ \cdots\ \alpha_n$`)` for annotating term t with attributes $\alpha_1, \ldots, \alpha_n$, are omitted when they annotate terms with zero attributes. Z3 ignores such annotations but SMT-LIBv2.6 specify that annotations must contain one or more attributes.

5. Introducing algebraic datatypes now follow SMT-LIBv2.6, where the datatypes are first defined with their respective arities followed by the declarations. We now always use the command `(declare-datatypes ((`$\delta_1\ k_1$`)` $\cdots$ `(`$\delta_n\ k_n$`))` `(`$d_1\ \cdots\ d_n$`))` for introducing $n > 0$ algebraic datatypes $\delta_1, \ldots, \delta_n$ with arities $k_1, \ldots, k_n$ and declarations $d_1, \ldots, d_n$ instead of using a Z3-specific command for single nullary datatypes, where datatypes and declarations were only separated by whitespace.

6. Pair-wise distinct, called `(distinct` $\cdots$`)`, is only encoded for at least two arguments instead of one.

4.2 Backend Support for Multiple SMT Solvers

Prior to this work, the Silicon backend supported only the Z3 solver. Fortunately, however, most of Z3's external behavior and output follows the SMT-LIB standard. As such, modifying Silicon to support multiple SMT solvers require only minor changes to the backend once Silicon has been modified to issue commands in the SMT-LIB command language.

We generalize the previous interface between Z3 and Silicon to instead be an abstract class named `ProverStdIO` that defines how Silicon interacts with other SMT-LIB conforming SMT solvers. Most of the previous methods are solver-agnostic due to SMT-LIB and can be reused directly. A notable exception is the setting of per query timeouts, whose method must now be overridden in subclasses that define the specific solvers. Additional issues with per query timeouts, specifically for cvc5, are discussed in Section 4.2.1.

To easily swap between different solvers in Silicon, we also add a new command line option to select which SMT solver should be used to discharge resulting proof obligations. The cvc5 solver can now be selected by running Silicon with the command `--prover=cvc5`. Furthermore, we generalize command line options to target all provers, where previously these were only targeting Z3. The naming of the new options follow the pattern `--prover<Option>` instead of `--z3<Option>`. We ensure backwards compatibility by fundamentally aliasing the old options as well as preserving any Z3-specific settings.

All the changes to the backend mean that we can now add support for a new SMT solver in Silicon in the following few steps:

1. Add a static config preamble defining the options that should be forwarded to the solver after instantiation (see e.g. `cvc5config.smt2`). Note that non-SMT-LIB conforming commands may be set as settings are often solver specific.
2. Implement a new object and a new class for the solver that extends the abstract `ProverStdIO` class defining solver details. Override the abstract methods `setTimeout` and `getProverPath` to define how per-check timeouts are set and the location of the solver binary, respectively.
3. If any, add reserved keywords used by the solver to `SmtlibNameGenerator.scala` to avoid Silicon-generated name clashes.
4. Add the new solver to the prover options in the command-line parser `Config.scala` and create a new case in the factory method `getProverStdIO` found in `Decider.scala`.
5. Add solver dependencies to `Silicon.scala`.

4.2.1 Supporting cvc5

After introducing backend support for multiple SMT solvers, adding support for cvc5 was relatively straightforward following the aforementioned steps. The static config preamble was based on the default options used for cvc5 by Boogie as well as translated options from the previous Z3 static config preamble in Silicon (`z3config.smt2`). We also experimented extensively with different settings defined by cvc5 to increase completeness and decrease runtime for the Silicon test suite but to no avail.

The only major issue faced in the implementation was that cvc5 does not support varying per query timeouts. After beginning a cvc5 instance using incremental solving, unlike Z3, we cannot change the per query timeout. It must either be disabled or set in the static config preamble as a global per

query timeout that remains for the entire lifetime of the cvc5 instance. Effectively, this means that per assertion and per check timeouts in Silicon, respectively through the `--assertTimeout` and `--checkTimeout` options, are ignored for the cvc5 solver. More importantly, it also implies that we are either forced to use Silicon's state saturation checks [2] with no timeout, or we have to disable these checks entirely.[3] Forcing state saturation checks may, however, result in some subset of queries taking significantly longer to prove or hanging indefinitely, even if the statements could otherwise be proved without these checks. In the latter case of disabling these checks, we may experience situations where lemmas are learned repeatedly in subsequent push and pop scopes as well as increased or indefinite query time. By default, we force state saturation checks with no timeout to keep options similar between all provers, but this can be changed through setting both `--proverSaturationTimeout` and `--proverSaturationTimeoutWeights` to any positive number. We explore the consequences of these forced or disabled state saturation checks in Chapter 5.

[2]In state saturation checks we issue additional `(check-sat)` commands to the prover in order to saturate its state and learn new lemmas early. Please refer to `https://github.com/viperproject/silicon/wiki/Performance-of-Silicon-on-long-running-examples-(e.g.-Nagini)` for details and earlier experiments with these checks in original Silicon using Z3.

[3]Likewise, this lack of varying per-query timeouts also affects Silicon's path feasibility checks. In path feasibility checks we query the SMT solver before branching to check whether the branching condition is already known to be true or false, and we can therefore avoid exploring an additional (and potentially expensive) path. By default, SMT-LIBv2.6 Silicon using cvc5 performs these checks with no timeout.

Chapter 5

Evaluation

We evaluate our implementation of an SMT-LIBv2.6 conformant Silicon using Z3 and cvc5 as SMT solvers against the original Silicon prior to this work with Z3-specific commands using Z3. Specifically, we test and benchmark the following “flavors” of Silicon:

1. Original Silicon with Z3-specific commands using Z3 vs. SMT-LIBv2.6 conformant Silicon using Z3
2. SMT-LIBv2.6 conformant Silicon using Z3 vs. using cvc5 with forced state saturation checks
3. SMT-LIBv2.6 conformant Silicon using cvc5 with forced state saturation checks vs. with disabled state saturation checks [1]

The purpose of the evaluation is twofold: (1) to assess whether Silicon's performance relies on any Z3-specific commands, and (2) to assess whether Silicon may benefit from supporting another SMT solver, specifically cvc5, in terms of runtime, soundness, or completeness. Moreover, we compare Silicon using cvc5 with forced and disabled state saturation checks in aid of purpose (2).

Both testing and benchmarking were performed on the standard Silicon test suite (`SiliconTests`) [2] with a timeout of 60 s and replicating every run 10 times. Further increasing the timeout to 5 minutes did not increase completeness for the tests but vastly increased the total runtime (from hours to days) and as such we only report results using a 60 s timeout. We

[1] A similar test and benchmark could be performed for path feasibility checks. We omit this due to time constraints.

[2] The standard test suite for Silicon consists of the following test directories in Silicon and Silver, respectively: `consistency`, `issue387`, `heuristics`, `symbExLogTests` and `all`, `quantifiedpermissions`, `quantifiedpredicates`, `quantifiedcombinations`, `wands`, `termination`, `examples`.

Table 5.1: Details of tools used in testing and benchmarking.

Tool	Version	Release date
cvc5	1.0.0	Apr 06, 2022
Java	11.0.13 LTS	Oct 19, 2021
Silicon (original)	1.1-SNAPSHOT edb5d079	Dec 07, 2021
Silicon (SMT-LIBv2.6)	1.1-SNAPSHOT a1badb18+@cvc5	Apr 21, 2022
Silver	SNAPSHOT 7228e714	Dec 02, 2021
Z3	4.8.7	Nov 20, 2019

did not experiment with increasing timeouts beyond 5 minutes as we believe user-friendliness, e.g. for frontend verifiers, would be severely impacted, and the verification would likely be interrupted regardless. Additionally, we include a short discussion of some simple arithmetic tests (`SimpleArithmeticTermSolverTests`) and some type tests (`SiliconBackendTypeTest`) defined in Silicon.

Our testing and benchmarking were performed on a 2021 MacBook Pro (Model Identifier MacbookPro18,3) with the Apple M1 Pro chip and 16 GB of memory running macOS Monterey 12.3.1. The details of the tools used are listed in Table 5.1.

5.1 Completeness and Soundness Testing

Before benchmarking, we tested the completeness and soundness of our implementation against the original Silicon with Z3. In Table 5.2 we list a high-level overview of our test results for the four different combinations of Silicon and SMT-solvers. Both the original Silicon and the SMT-LIBv2.6 Silicon were able to pass all tests as expected.

Swapping the SMT-solver from Z3 to cvc5, however, resulted in a significant amount of tests failing or timing out. The discrepancy between cancelled tests for Silicon using Z3 and cvc5 is explained by the fact that the remainder (19 and 18) of these cancelled tests failed or timed out for SMT-LIBv2.6 using cvc5 with forced and disabled state saturation checks instead. After close inspection of the failed tests, we present their reasons for failure in Table 5.3. We note that using cvc5 results in identical failed tests for both forced and disabled state saturation checks except for `issues/silicon/0512.vpr` [3]. This test timed out when forcing checks but is imprecise and failed to verify when checks were disabled. Unfortunately, Silicon using cvc5 is suspected to be unsound for six tests with no obvious common cause. We explore this unsoundness in Section 6.1. Surprisingly, however, using cvc5 resulted in 10

[3] Paths to test files are relative to the Silver and Silicon test directories.

Table 5.2: Overview of results from the standard Silicon test suite. Note that failed tests do not include those that timed out. Cancelled tests indicate that the test passed, but there were annotations for unexpected or missing input (or, in general, the test failed with some specific unrelated error type). SSCs is an abbreviation for state saturation checks.

	Original Silicon	SMT-LIBv2.6 Silicon		
	Z3	Z3	cvc5 (forced SSCs)	cvc5 (disabled SSCs)
Passed	956	956	868	869
Failed	0	0	39	40
Timed out	0	0	68	65
Cancelled	80	80	61	62

Table 5.3: Reasons for failure in the standard Silicon test suite. Unsound denotes that the verification was unsound. Precise denotes that Silicon was able to verify the program correctly and may indicate that some test annotations are incorrect. Imprecise denotes that Silicon was not able to verify the program.

	SMT-LIBv2.6 Silicon	
	cvc5 (forced SSCs)	cvc5 (disabled SSCs)
Unsound	6	6
Precise	10	10
Imprecise	23	24

test programs being verified correctly that Z3-based Silicon failed to verify. These tests were incorrectly marked as failed, often due to the fact that some tests rely on the SMT-solver not being able to prove assertions that actually hold. Similarly, some tests rely on and are annotated with some unexpected (and incorrect) output of the SMT-solver. We elaborate on this increased completeness compared to Silicon using Z3 in Section 6.2. The remaining 23 (and 24) tests do not indicate any unsoundness; cvc5 was simply not able to prove entailment. The majority of these 23 (24) failed tests involve universal quantification, often in the form of quantified permissions or predicates.

Finally, all simple arithmetic tests passed for all evaluated combinations of Silicon with Z3 and cvc5. However, two type tests failed with SMT-LIBv2.6 Silicon using cvc5, namely `typeCombinationFail` and `fieldTypeSuccess`. This is due to cvc5 only having experimental support for floating points with sizes other than 8/24 (Float32) and 11/53 (Float64) while the aforementioned two tests use floating points of size 6/23. Enabling the experimental FP solver allows these tests to pass but due to known issues with the solver, we chose not to enable it by default in Silicon's cvc5 instances. Users may, as previously, forward any options to cvc5 via the command line options `--proverArgs` and `--proverConfigArgs` in Silicon, including enabling this experimental FP solver.

5.2 Benchmarking

In the following, we benchmark the runtime performance of our implementation on the standard Silicon test suite in the three previously mentioned settings, respectively: (1) Original Silicon vs. SMT-LIBv2.6 Silicon using Z3, (2) SMT-LIBv2.6 Silicon using Z3 vs. cvc5, and (3) SMT-LIBv2.6 Silicon using cvc5 with forced vs. disabled state saturation checks. We discard failed, cancelled and ignored tests. Benchmarks per test directory in the standard Silicon test suite can be found in Appendix A.

5.2.1 Original Against SMT-LIBv2.6 Flavored Silicon

A runtime comparison between original Silicon and SMT-LIBv2.6 Silicon using Z3 is shown in Figure 5.1. Both versions showed very similar performance, as expected due to their minor differences, with a mean runtime of (0.704 ± 0.026) s across all tests in original Silicon and (0.714 ± 0.029) s in SMT-LIBv2.6 Silicon.

We, however, noticed two outliers where SMT-LIBv2.6 was significantly slower. The first and most extreme outlier is `examples_paper/list_insert.vpr` with a mean runtime of (26.428 ± 3.001) s in SMT-LIBv2.6 Silicon compared to just (2.061 ± 0.100) s in original Silicon. Upon examination, we were not able to explain this 12-fold increase in runtime by the content of the test program and its new translation to the SMT-LIB command language. Furthermore, we were not able to reproduce this difference in runtime when rerunning the test. Here we found no significant difference and as such we assume this to be a spurious outlier. The other noticeable outlier is `third_party/testTranspose.vpr` with a mean runtime of (2.553 ± 0.047) s in SMT-LIBv2.6 Silicon compared to (1.429 ± 0.018) s in original Silicon. Again this significant difference was not reproducible or explained by the new command translation.

Discarding these outliers, we have a mean runtime of (0.701 ± 0.026) s across all tests in original Silicon and (0.685 ± 0.026) s in SMT-LIBv2.6 Silicon. Overall, it is likely that there is no significant difference in runtime between the two versions.

5.2.2 Z3- Against cvc5-Based Silicon

Similarly, Figure 5.2 shows a runtime comparison between SMT-LIBv2.6 Silicon using Z3 and cvc5 with forced state saturation checks. The mean runtime across all tests when using Z3 was, as before, (0.714 ± 0.029) s and (5.030 ± 0.021) s when using cvc5. This major difference was due to 68 tests timing out after 60 s with cvc5. Excluding timeouts, we instead have a mean of (0.622 ± 0.024) s and (0.710 ± 0.021) s for Z3 and cvc5, respectively.

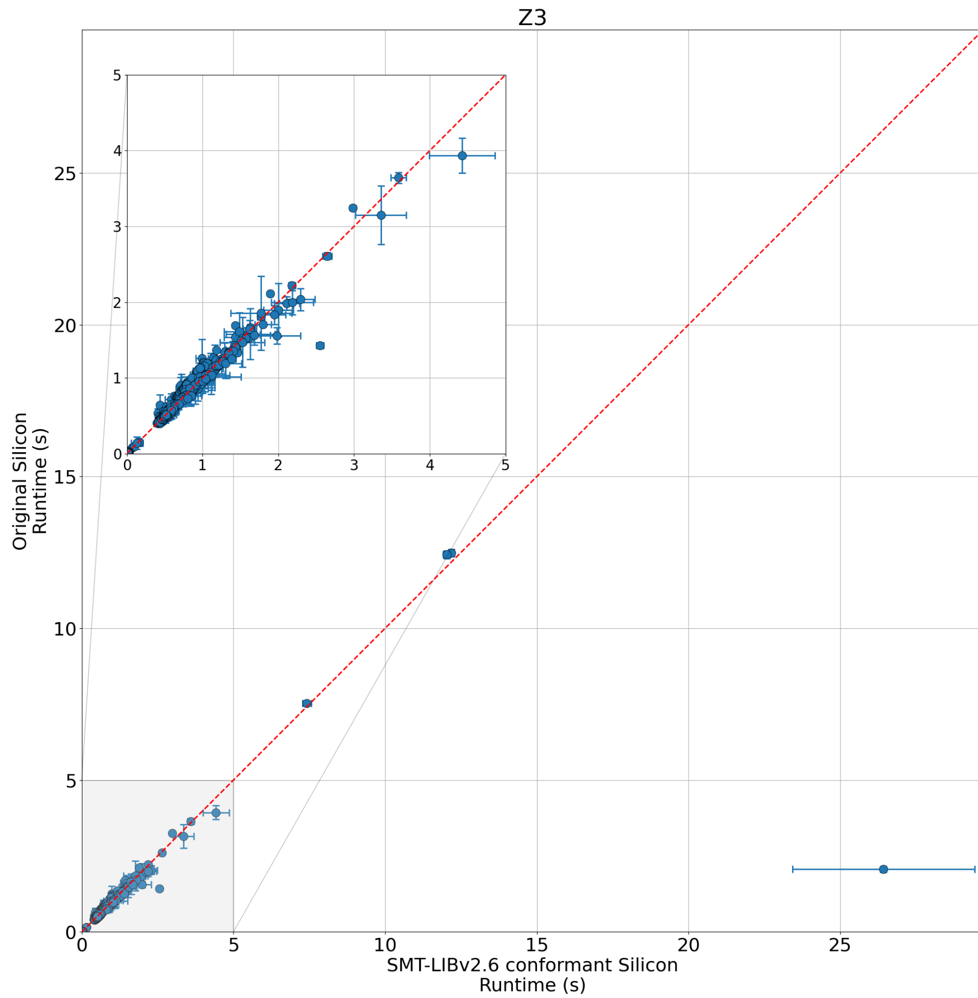

Figure 5.1: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis). Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

The majority of tests timing out involve universal quantifiers, especially quantified permissions. Particularly, in just the Silver test directory `quantifiedpermissions`, 45 out of 140 (32.14 %) tests timed out. These quantified permission-related timeouts therefore accounted for 66.18 % of all those tests that timed out when using cvc5 with forced state saturation checks, even though they only made up 140 out of 1036 (13.51 %) of the total number of tests. Comparatively, in the more general tests in the Silver test directory `all`, only 13 out of 669 (1.94 %) tests timed out. Tests with frequent use of universal quantifiers experienced a greater-than-average amount of timeouts or failures compared to those without. Statistics and benchmarks for each test directory can be found in Appendix A.

Moreover, we note that passing `quantifiedpermissions` tests take significantly (often several magnitudes) longer to verify when using cvc5 compared to Z3. This is not a trend that can be ascribed to all passed tests with universal quantifiers but seem to be specific to those with quantified permissions.

In general, tests that were able to verify in less than 0.8 s with both Z3 and cvc5, verify slightly faster in cvc5. For these tests cvc5 provides up to a 1.75x speedup over Z3, generally experiencing higher speedups for tests with shorter verification times. Tests that take longer than 0.8 s to verify were usually significantly faster with Z3.

5.2.3 To Force or Not To Force State Saturation Checks

The final benchmark is depicted in Figure 5.3, where we compare SMT-LIBv2.6 Silicon using cvc5 when forcing and disabling state saturation checks. The mean runtime across all tests, timeouts included, were (5.030 ± 0.021) s for forced and (4.925 ± 0.020) s for disabled checks. Excluding timeouts, the means were reduced to (0.706 ± 0.021) s and (0.764 ± 0.020) s for forced and disabled checks, respectively

We observed a trade-off between forcing and disabling these state saturation checks. Both configurations introduced some impreciseness in the verification that was not present in the other. For example, with forced checks we passed the tests `issues/carbon/0196.vpr` and `issues/silicon/0362.vpr` but with disabled checks, they timed out. Comparably, with disabled checks we were able to pass `issues/issue_0081b.vpr`, `issues/issue_0170.vpr`, and `sequences/mergesort.vpr`, but the three tests timed out when checks were forced. By inspection, we were not able to exactly determine what triggers this behavior, but we suspect matching loops were introduced in each case.

Overall, forcing state saturation checks significantly decreases runtime for tests that take longer than $\sim$5 s to verify and tests taking longer than $\sim$5 s

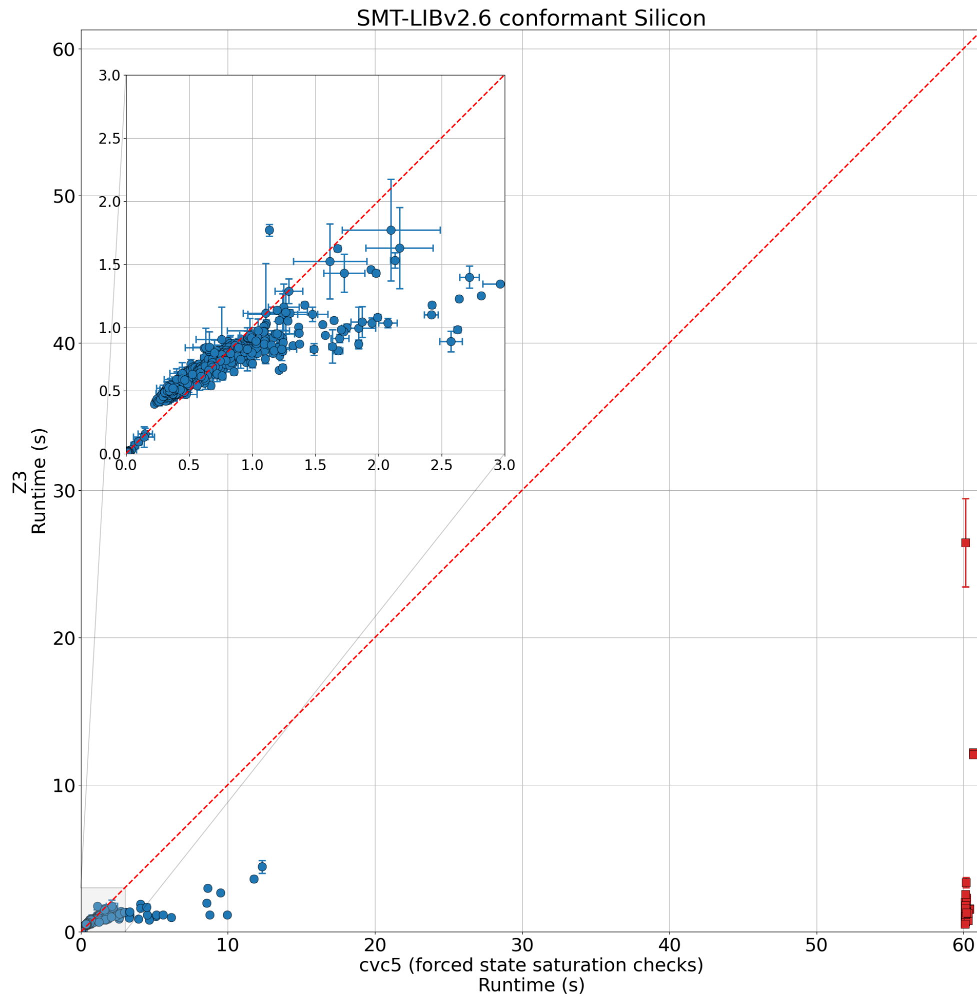

Figure 5.2: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis). Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

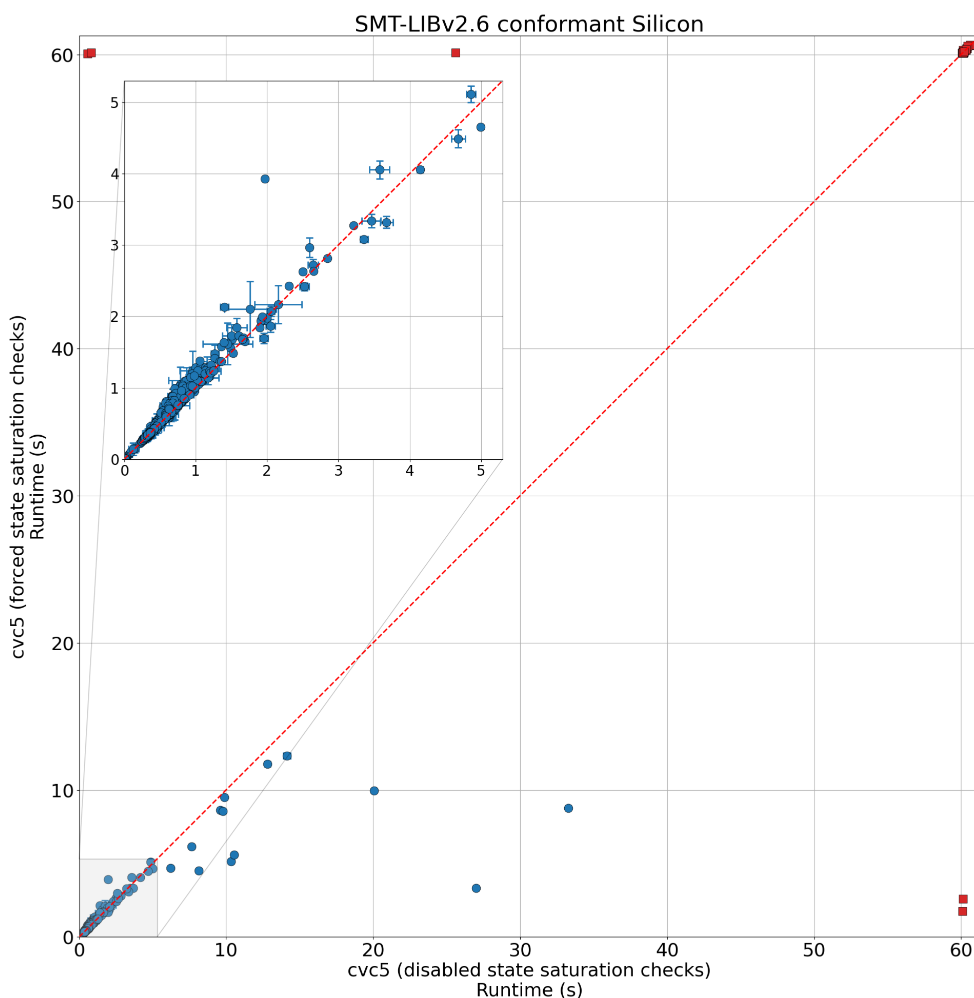

Figure 5.3: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

to verify generally only see a minor increase in runtime. Forcing checks, however, comes at the cost of potentially introducing more timeouts. Users willing to sacrifice general runtime in longer verification tasks may opt to disable state saturation checks when using cvc5 through the command line options mentioned in Section 4.2.1 for improved completeness.

Chapter 6

Discussion

In this section, we highlight and elaborate on some interesting results found during testing and benchmarking.

6.1 Unsoundness With cvc5

In our testing of SMT-LIBv2.6 Silicon using cvc5 with forced and disabled state saturation checks, we discovered six tests on which the two Silicon flavors were unsound.[1] Three of these tests relate to access permissions, where the Silicon flavors are expected to fail with insufficient permission but did not. One is related to indexing a sequence using two nested universal quantifiers, where an index exceeding sequence length is not detected. Another is related to cardinality for multisets, where the assertions should fail due to multisets having no strong axioms, but verification succeeded instead. In the following, we will briefly elaborate on the last test concerning postconditions in functions.

A minimal extract of the test showing unsound behavior in the discussed cvc5-based Silicon flavors is depicted in Listing 6.1. The test declares a field called `x` of type `Int` and function called `postFunction2` that takes as argument a non-null reference with permission to field `x`. The function simply returns 1. We expect not to be able to prove the postcondition in line 4 since it is not guaranteed by the function. However, the discussed Silicon flavors verify this in contrast to Z3-based Silicon flavors where verification fails.

The observed unsoundness appears to be caused by the postcondition referencing something other than its result. We illustrate this by a minimal unsound example in Listing 6.2 that incorrectly verifies for the discussed cvc5-

[1] The six unsound tests are (in the order as mentioned): `issues/carbon/0271.vpr`, `new_syntax/QPFields.vpr`, `new_syntax/QPPredicates.vpr`, `issues/silver/0175.vpr`, `multisets/multisets.vpr`, and `issues/carbon/0069.vpr`.

```
0 field x: Int
1
2 function postFunction2(this: Ref): Int
3   requires this != null && acc(this.x)
4   ensures this.x == 0
5 {
6   1
7 }
```

Listing 6.1: Minimal extract of unsoundness discovered in test `issues/carbon/0069.vpr` for SMT-LIBv2.6 Silicon using cvc5. This behavior was observed for both forced and disabled state saturation checks.

```
0 function unsound(this: Int): Int
1   requires this == 0
2   ensures false
3 {
4   1
5 }
```

Listing 6.2: Minimal example of unsoundness discovered in SMT-LIBv2.6 Silicon using cvc5.

based Silicon flavors but correctly fails for Z3 flavors. Relevant sections of the corresponding SMT-LIB script generated by Silicon as well as responses by the SMT solver are shown in Listing 6.3. We omit large amounts of SMT-LIB commands such as those defined in the static preambles as well as some well-definedness assertions. In this case, we see that the SMT solver (here cvc5) responds to the last `(check-sat)` command with `unsat` indicating that the formula is valid. As such, the verification (incorrectly) succeeds and the postcondition must (incorrectly) hold. If we instead use Z3 flavors of Silicon, Z3 responds `unknown` indicating correctly that the postcondition might not hold. Similar examples can be constructed for the other five unsound tests we discovered. Further investigation is required to determine whether these unsound examples are caused by unsoundness in cvc5 or whether they only emerge in the specific Silicon flavors.

6.2 Completeness and Improper Test Annotations

During our testing we found 10 tests in the standard Silicon test suite, where cvc5-based Silicon flavors were able to correctly verify the test programs but Z3-based flavors were not.[2] The results of these tests were, however, incorrectly marked as failed since they contain annotations that rely on Z3 not

[2]The tests are: `basic/quantifiers.vpr`, `functions/heap_dependent_triggers.vpr`, `issues/silicon/0005b.vpr`, `issues/silicon/0183.vpr`, `issues/silicon/0491.vpr`, `issues/issue_0064.vpr`, `issues/issue_0147.vpr`, `misc/triggers_field_deref.vpr`, `misc/unbounded.vpr`, and `sets/nonnull.vpr`.

```
0  ...
1  (declare-fun unsound ($Snap Int) Int)
2  ...
3  ; ---------- FUNCTION unsound----------
4  (declare-fun this@0@00 () Int)
5  (declare-fun result@1@00 () Int)
6  ...
7  ; ----- Verification of function body and postcondition -----
8  (push) ; 1
9  (assert (= s@$ $Snap.unit))
10 (assert (= this@0@00 0))
11 ; State saturation: after contract
12 (check-sat)
13 ; unsat
14 (assert (= result@1@00 1))
15 (push) ; 2
16 (assert (not false))
17 (check-sat)
18 ; unsat
```

Listing 6.3: Extract of the SMT-LIB script generated by SMT-LIBv2.6 Silicon using cvc5 with forced state saturation checks when verifying the example in Listing 6.2.

being able to prove assertions that actually hold or where Z3 has unexpected output. Several of these tests are related to open issues in the original flavor of Silicon such as issue #80 [3], #183 [4], and #204 [5]. Using cvc5 as the backend solver of SMT-LIBv2.6 Silicon fixes both issue #183 and #204 as well as partially fixes problems mentioned in issue #80.

In Listing 6.4 we show an extract of issue #183 involving non-linear real arithmetic with permissions that fails to verify with Z3-based Silicon flavors but succeeds with cvc5-based flavors. The test annotations rely on the SMT-solver not being able to prove the assertion even if it actually holds. The test is thus marked as failed even when cvc5-based Silicon flavors correctly verify the program.

Finally, we remark that Silicon has always been developed with Z3 in mind and the entire Silicon test suite is heavily biased towards Z3. All tests pass and are verifiable in a reasonable amount of time with Z3-based Silicon flavors by design. Front ends are developed such that programs can be verified with Z3 as the backend SMT solver. Consequently, it is difficult to utilize the existing test suite to illustrate scenarios where Silicon may benefit from other SMT solvers such as cvc5 (except for improperly annotated tests as seen here). Highlighting weak spots, such as reduced completeness or increased runtime for some inputs, is much easier.

[3] https://github.com/viperproject/silicon/issues/80
[4] https://github.com/viperproject/silicon/issues/183
[5] https://github.com/viperproject/silicon/issues/204

```
0 method foo_fail_gist(self: Ref, rd: Perm)
1   requires none < rd && rd < write
2 {
3   //:: UnexpectedOutput(assert.failed:assertion.false, /
      Silicon/issue/183/)
4   //:: UnexpectedOutput(assert.failed:assertion.false, /Carbon
      /issue/83/)
5   assert none < rd * rd
6 }
```

Listing 6.4: Extract of issue #183 that can be verified with cvc5-based Silicon flavors.

Chapter 7

Conclusion

We presented a new flavor of Silicon introducing SMT-LIBv2.6 conformance and backend support for multiple SMT solvers with equal performance to the original Silicon flavor. To do so, we identified Z3-specific SMT commands that Silicon previously relied on and implemented new SMT-LIBv2.6 conformant Silicon with a generalized SMT-solver interface. Subsequently, we were able to introduce support for the cvc5 solver yielding increased completeness for 10 tests in the standard Silicon test suite compared to Z3 but also unsoundness in six tests. The cvc5-based flavors of Silicon were imprecise and failed 23 tests when state saturation checks were forced (and 24 when disabled) that otherwise passed with Z3-based flavors. Likewise, we observed 68 and 65 tests timing out for cvc5-based flavors with forced and disabled state saturation checks, respectively, indicating a significant decrease in completeness relative to Z3-based flavors for the current test suite, especially in tests using quantified permissions. Finally, we performed comprehensive benchmarking of the different flavors of Silicon that we introduced, and showed that SMT-LIBv2.6 conformance did not hinder performance compared to the original flavor of Silicon. While the cvc5 solver may not replace Z3 as the default backend of Silicon yet, it may still prove to be a useful alternative.

7.1 Future Work

Future work might explore several directions focused on the backend SMT solvers used by Viper.

Additional benchmarking for cvc5-based Silicon flavors may be interesting. For example, in this work we have ignored Silicon's path feasibility checks and instead focused on state saturation checks. These checks may have equally interesting trade-offs such as the one seen for state saturation checks, where runtime in large verification tasks can be sacrificed for improved com-

pleteness. Likewise, it may prove fruitful to experiment with cvc5's syntax-guided quantifier instantiation technique [15] that uses a counterexample-guided approach to synthesize terms for quantifier instantiation with its SyGuS solver.

Introducing support for cvc5 in the VC-generation-based verifier Carbon can also be a worthwhile endeavor with the same benefits as seen here in Silicon. This support should effectively come for free once the Boogie version used by Carbon is upgraded to one of the later versions that include (experimental) support for cvc5 and Yices 2.

Lastly, exploring heuristics to swap between solvers depending on input is an interesting idea that could help increase completeness and efficiency if the solvers have complementing strengths and weaknesses.

Appendix A

Per Test Directory Benchmarks And Statistics

A.1 Statistics Per Silicon Flavor

Statistics of test results per test directory are shown in Table A.1, Table A.2, Table A.3, and Table A.4 for original Silicon using Z3, SMT-LIBv2.6 Silicon using Z3, SMT-LIBv2.6 Silicon using cvc5 with forced state saturation checks, and SMT-LIBv2.6 Silicon using cvc5 with disabled state saturation checks, respectively.

Table A.1: Overview of test results per test directory for original Silicon using Z3.

	Passed	Failed	Timeout	Cancelled	Total
`all`	621	0	0	48	669
`consistency`	1	0	0	0	1
`examples`	13	0	0	0	13
`heuristics`	0	0	0	1	1
`issue387`	15	0	0	0	15
`quantifiedcombinations`	9	0	0	0	9
`quantifiedpermissions`	126	0	0	14	140
`quantifiedpredicates`	22	0	0	4	26
`symbExLogTests`	10	0	0	0	10
`termination`	28	0	0	1	29
`wands`	111	0	0	12	123

A.2 Original Against SMTLIBv2.6 Flavored Silicon

Per test directory benchmarks are shown in the following figures: Figure A.1, Figure A.2, Figure A.3, Figure A.4, Figure A.5, Figure A.6, Fig-

Table A.2: Overview of test results per test directory for SMT-LIBv2.6 Silicon using Z3.

	Passed	Failed	Timeout	Cancelled	Total
`all`	621	0	0	48	669
`consistency`	1	0	0	0	1
`examples`	13	0	0	0	13
`heuristics`	0	0	0	1	1
`issue387`	15	0	0	0	15
`quantifiedcombinations`	9	0	0	0	9
`quantifiedpermissions`	126	0	0	14	140
`quantifiedpredicates`	22	0	0	4	26
`symbExLogTests`	10	0	0	0	10
`termination`	28	0	0	1	29
`wands`	111	0	0	12	123

Table A.3: Overview of test results per test directory for SMT-LIBv2.6 Silicon using cvc5 with forced state saturation checks.

	Passed	Failed	Timeout	Cancelled	Total
`all`	593	22	13	41	669
`consistency`	1	0	0	0	1
`examples`	9	2	2	0	13
`heuristics`	0	0	0	1	1
`issue387`	14	1	0	0	15
`quantifiedcombinations`	7	1	1	0	9
`quantifiedpermissions`	81	8	45	6	140
`quantifiedpredicates`	19	3	2	2	26
`symbExLogTests`	10	0	0	0	10
`termination`	26	0	2	1	29
`wands`	108	2	3	10	123

ure A.7, Figure A.8, Figure A.9, Figure A.10, and Figure A.11. Note that some plots are empty due to having no passed or timed out tests. The zoomed in region shows the region between the 10 % and 90 % quantiles.

A.3 Z3- Against cvc5-Based Silicon

Per test directory benchmarks are shown in the following figures: Figure A.12, Figure A.13, Figure A.14, Figure A.15, Figure A.16, Figure A.17, Figure A.18, Figure A.19, Figure A.20, Figure A.21, and Figure A.22. Note that some plots are empty due to having no passed or timed out tests. The zoomed in region shows the region between the 10 % and 90 % quantiles.

Table A.4: Overview of test results per test directory for SMT-LIBv2.6 Silicon using cvc5 with disabled state saturation checks.

	Passed	Failed	Timeout	Cancelled	Total
`all`	591	23	13	42	669
`consistency`	1	0	0	0	1
`examples`	9	2	2	0	13
`heuristics`	0	0	0	1	1
`issue387`	14	1	0	0	15
`quantifiedcombinations`	7	1	1	0	9
`quantifiedpermissions`	84	8	42	6	140
`quantifiedpredicates`	19	3	2	2	26
`symbExLogTests`	10	0	0	0	10
`termination`	26	0	2	1	29
`wands`	108	2	3	10	123

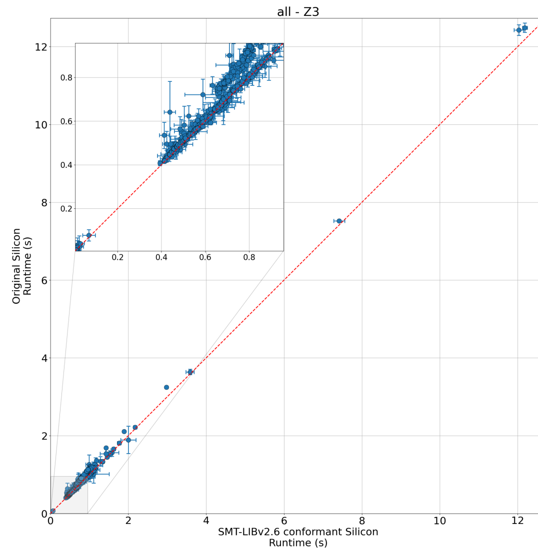

Figure A.1: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `all` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

A.4 To Force or Not To Force State Saturation Checks

Per test directory benchmarks are shown in the following figures: Figure A.23, Figure A.24, Figure A.25, Figure A.26, Figure A.27, Figure A.28, Figure A.29, Figure A.30, Figure A.31, Figure A.32, and Figure A.33. Note that some plots are empty due to having no passed or timed out tests. The zoomed in region shows the region between the 10 % and 90 % quantiles.

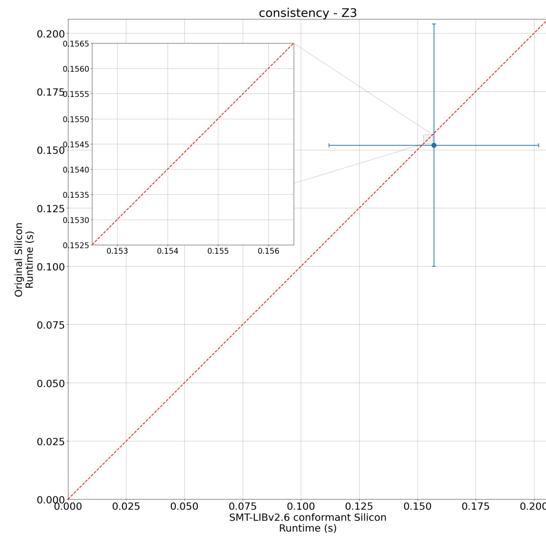

Figure A.2: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `consistency` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

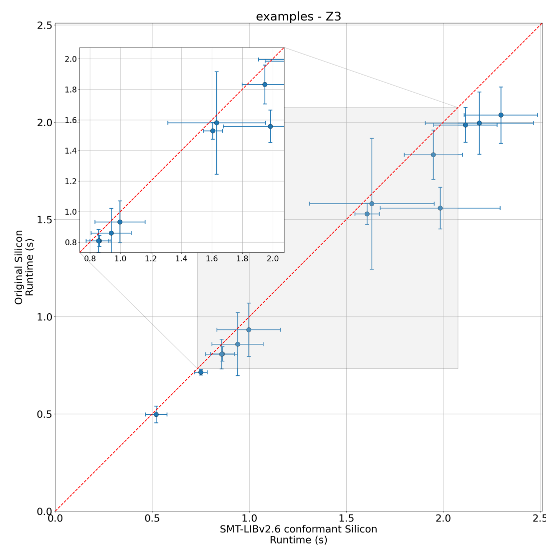

Figure A.3: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `examples` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

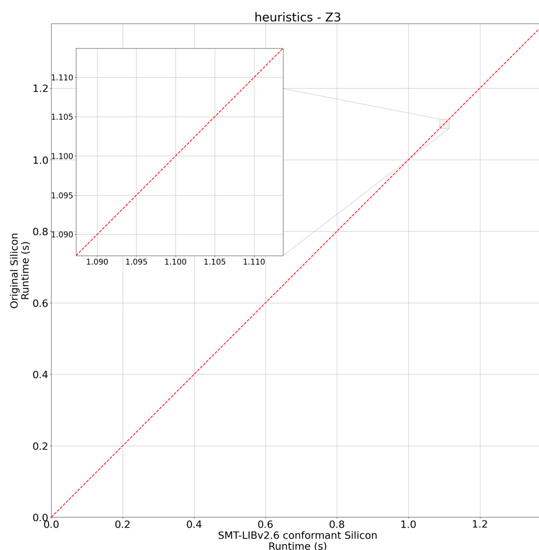

Figure A.4: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `heuristics` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

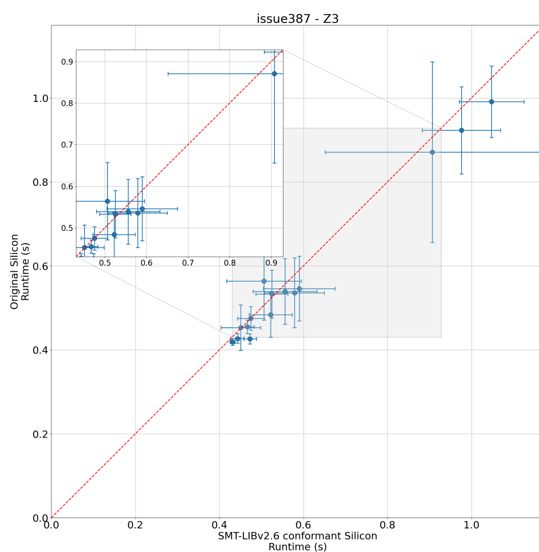

Figure A.5: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `issue387` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

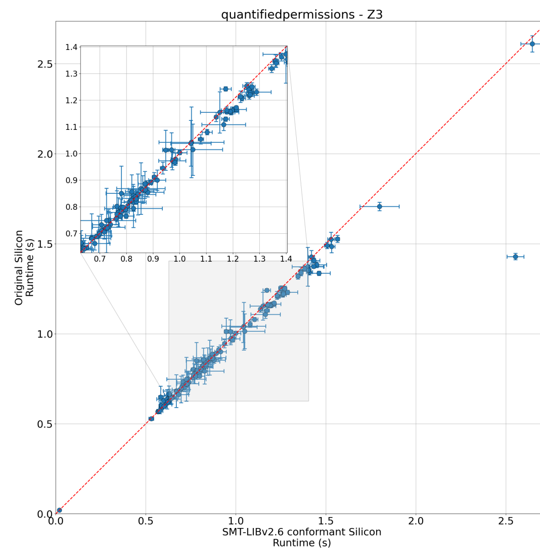

Figure A.6: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `quantifiedpermissions` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

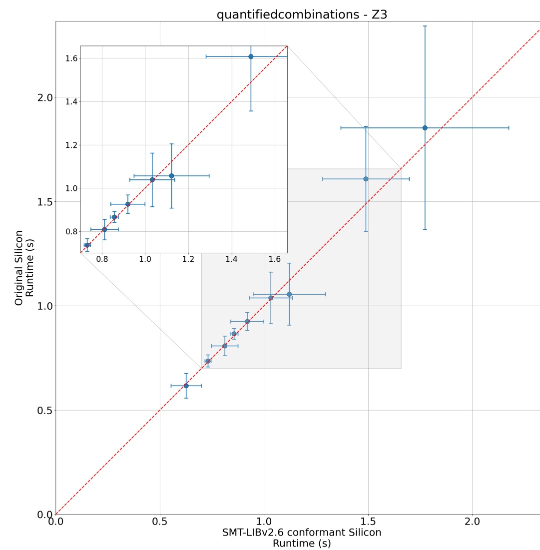

Figure A.7: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `quantifiedcombinations` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

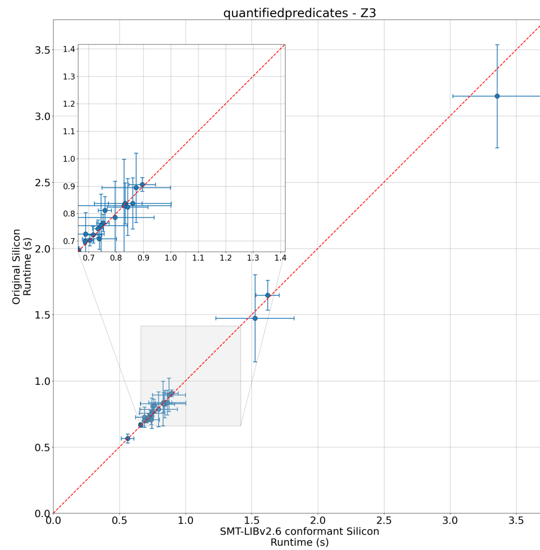

Figure A.8: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `quantifiedpredicates` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

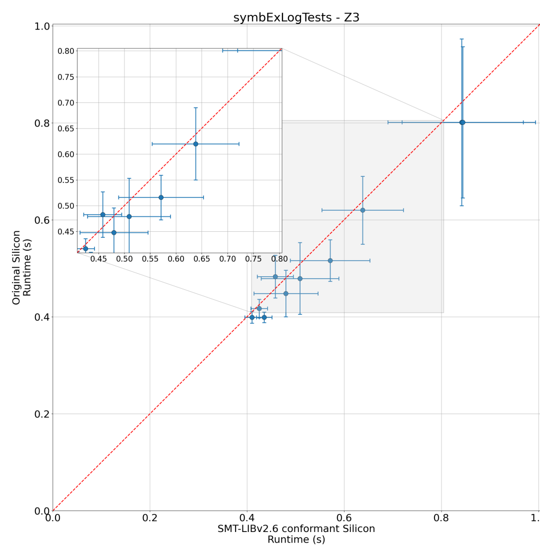

Figure A.9: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `symbExLogTests` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

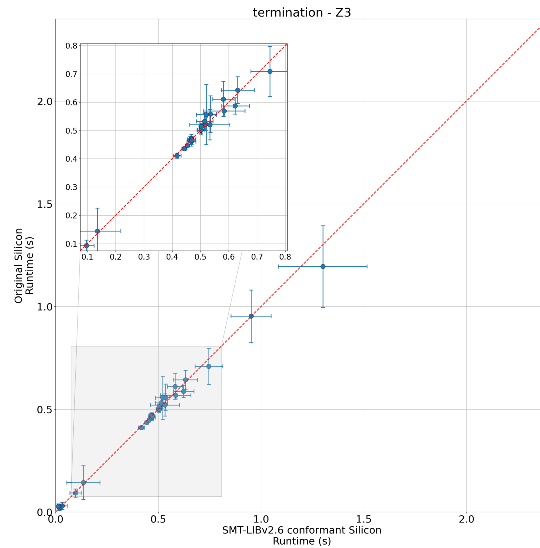

Figure A.10: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `termination` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

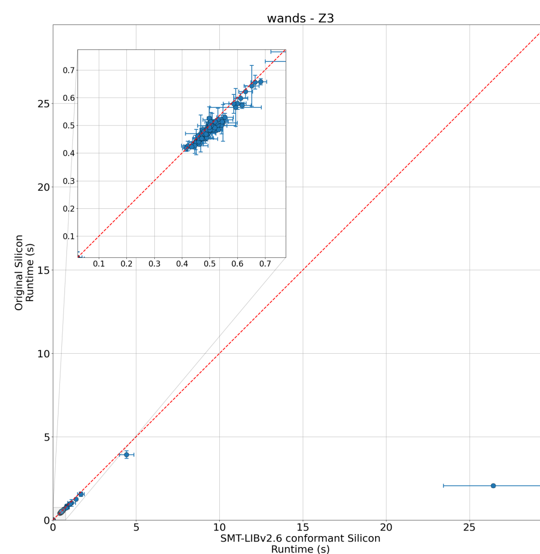

Figure A.11: Runtime comparison between original Silicon using Z3 (y-axis) and SMT-LIBv2.6 Silicon using Z3 (x-axis) for the `wands` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

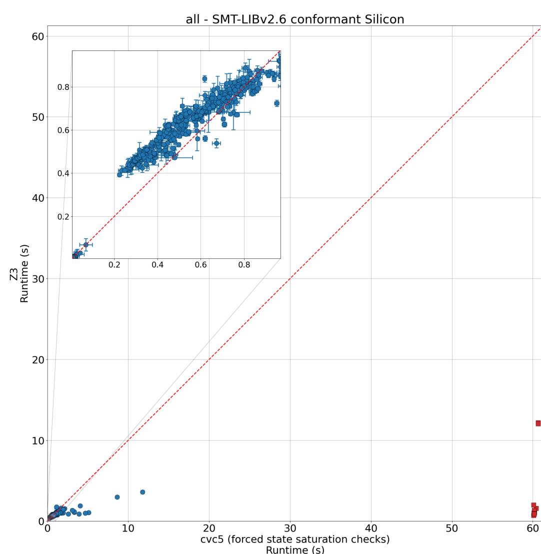

Figure A.12: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `all` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

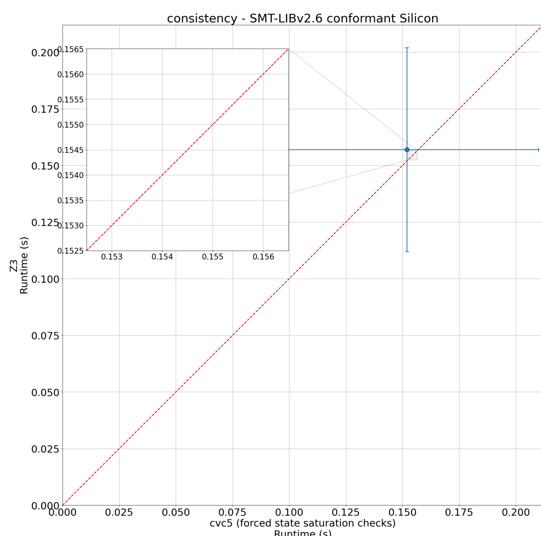

Figure A.13: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `consistency` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

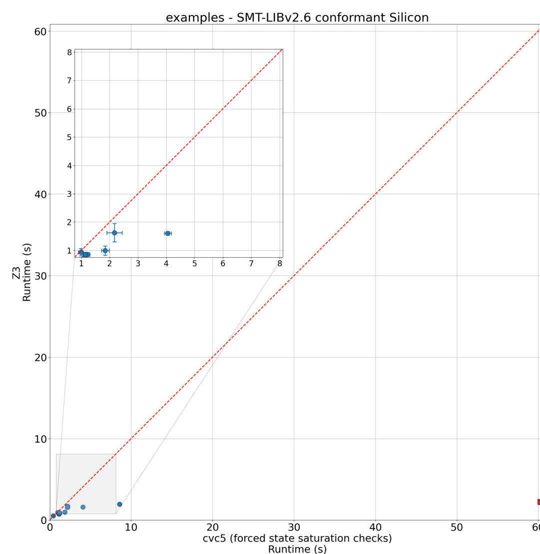

Figure A.14: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `examples` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

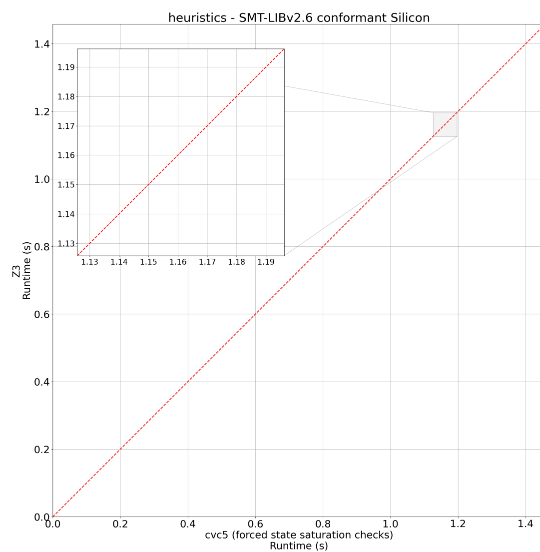

Figure A.15: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `heuristics` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

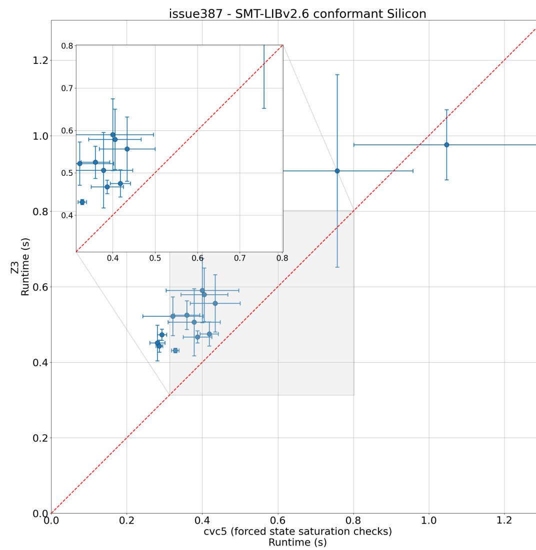

Figure A.16: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `issue387` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

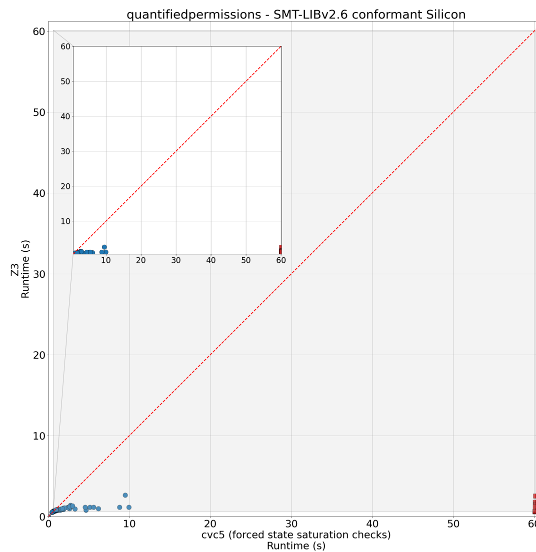

Figure A.17: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `quantifiedpermissions` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

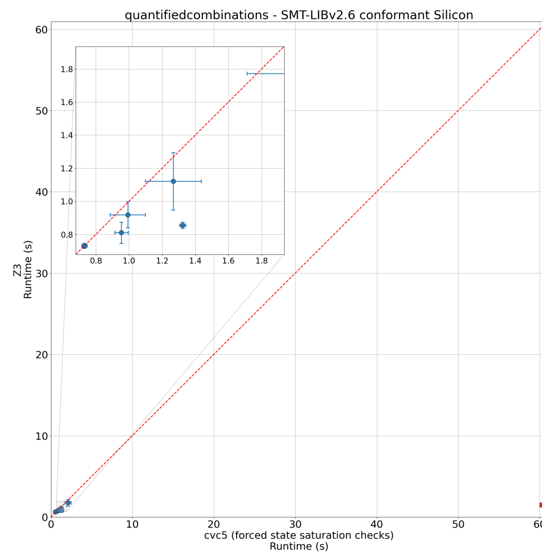

Figure A.18: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `quantifiedcombinations` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

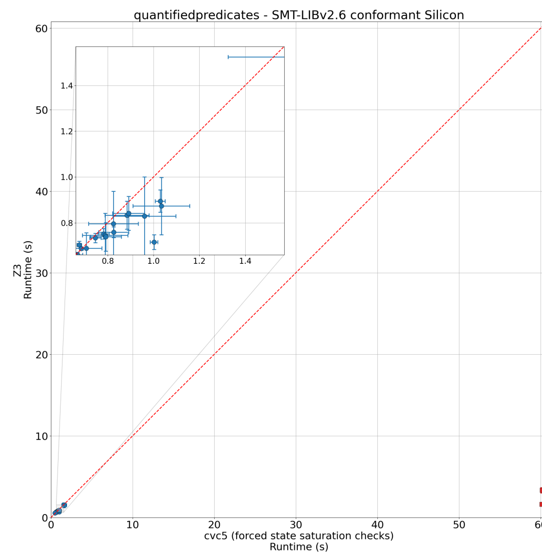

Figure A.19: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `quantifiedpredicates` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

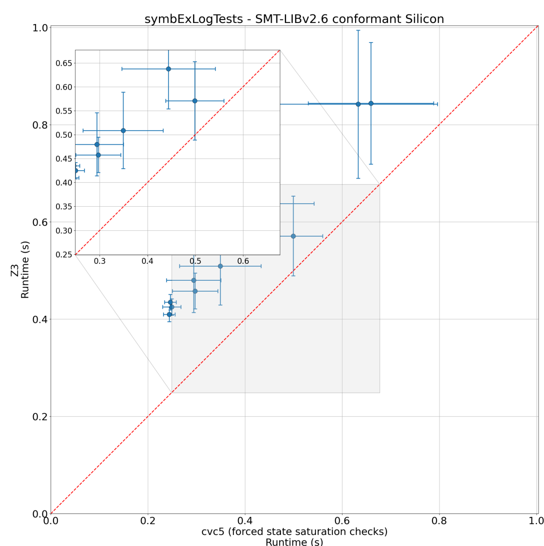

Figure A.20: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `symbExLogTests` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

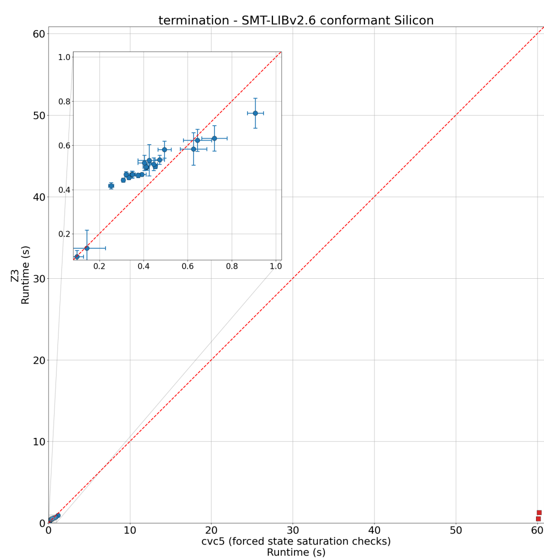

Figure A.21: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `termination` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

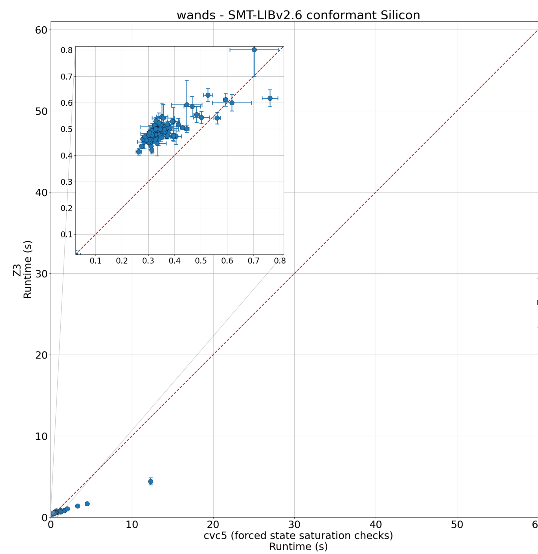

Figure A.22: Runtime comparison between SMT-LIBv2.6 Silicon using Z3 (y-axis) and using cvc5 with forced state saturation checks (x-axis) for the `wands` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

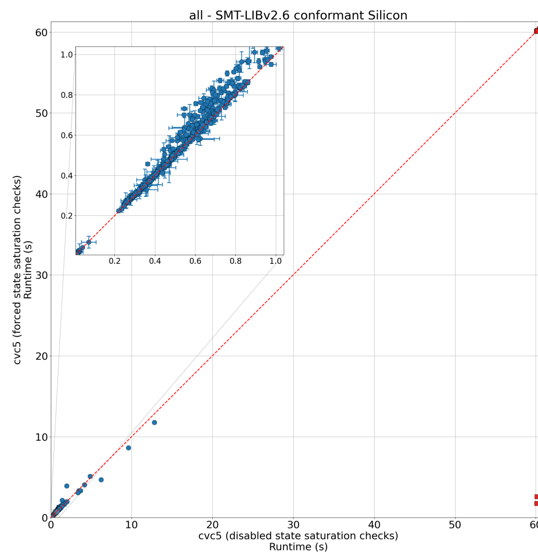

Figure A.23: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `all` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

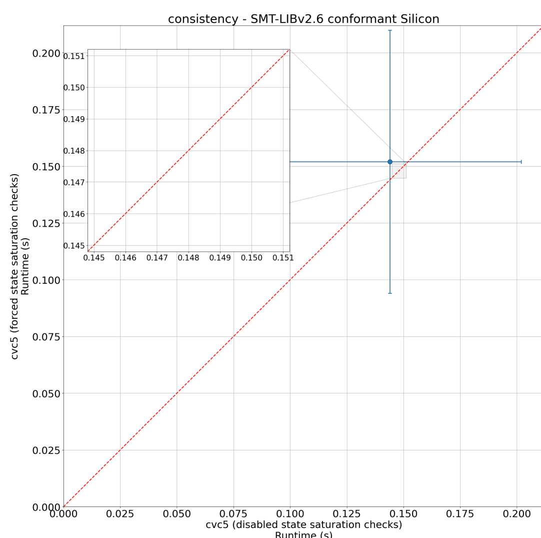

Figure A.24: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `consistency` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

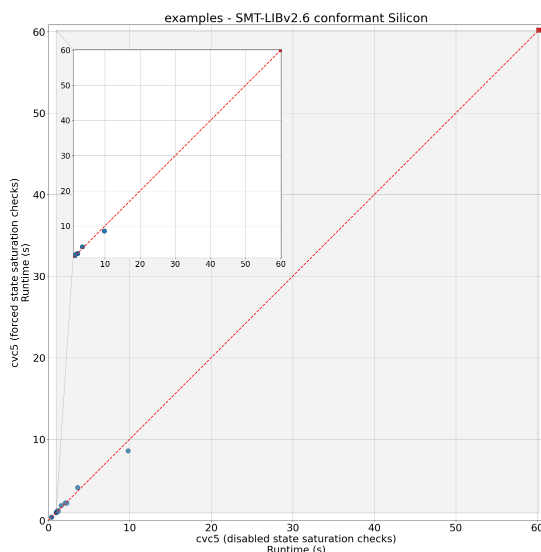

Figure A.25: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `examples` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

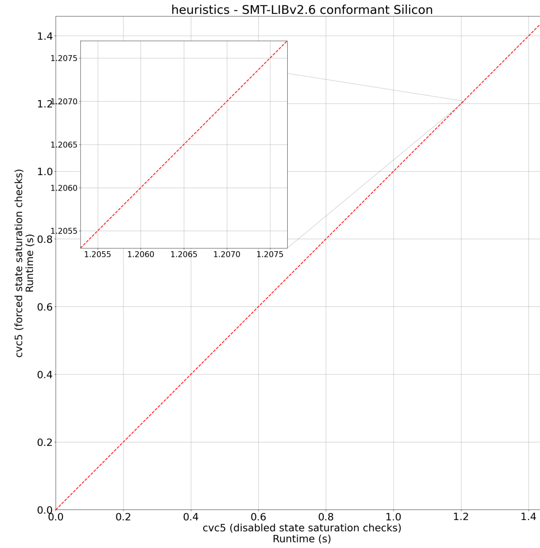

Figure A.26: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `heuristics` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

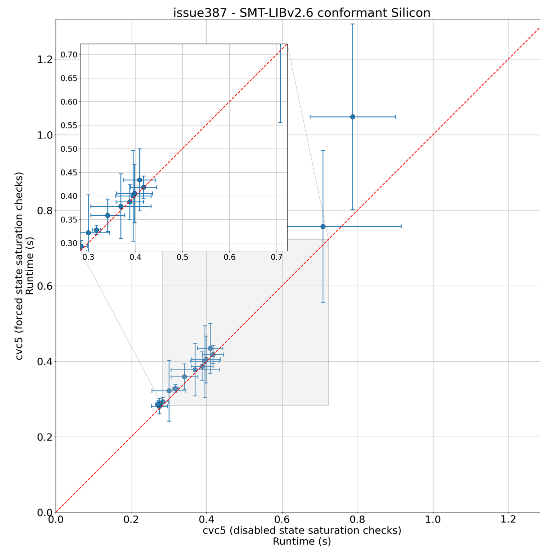

Figure A.27: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `issue387` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

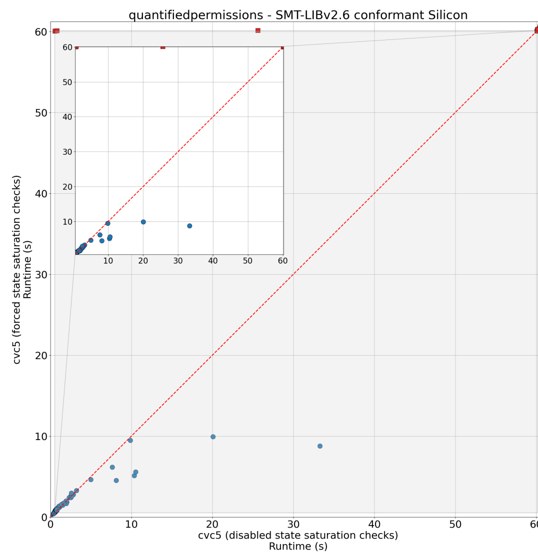

Figure A.28: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `quantifiedpermissions` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

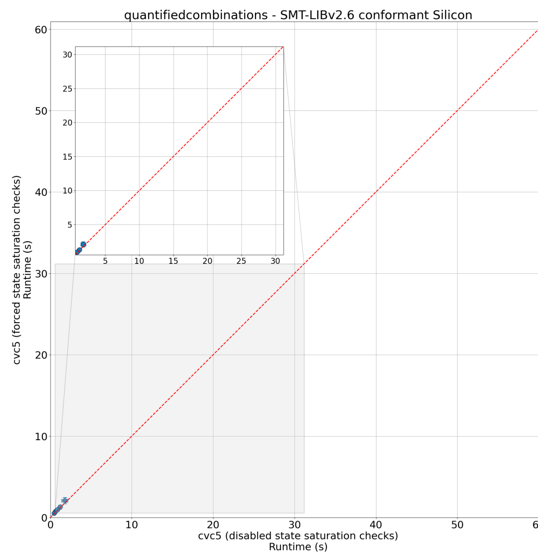

Figure A.29: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `quantifiedcombinations` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

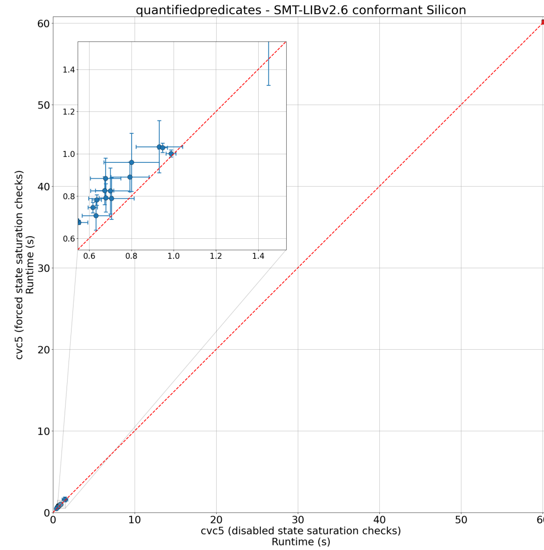

Figure A.30: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `quantifiedpredicates` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

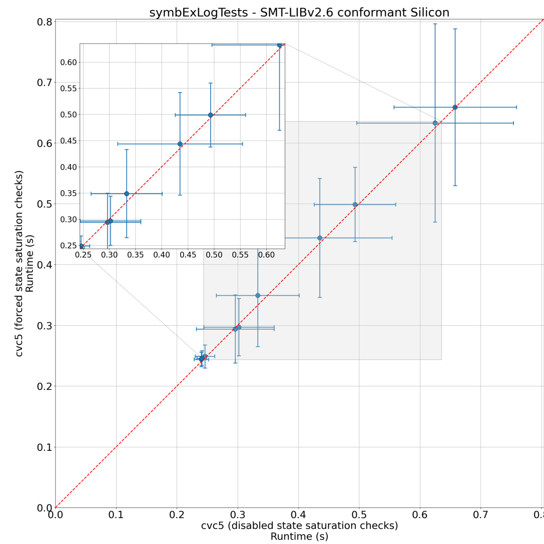

Figure A.31: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `symbExLogTests` test directory. Data points are means ± standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

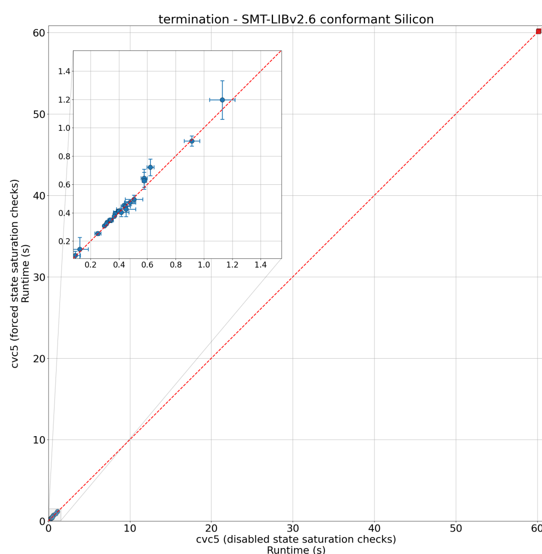

Figure A.32: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `termination` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

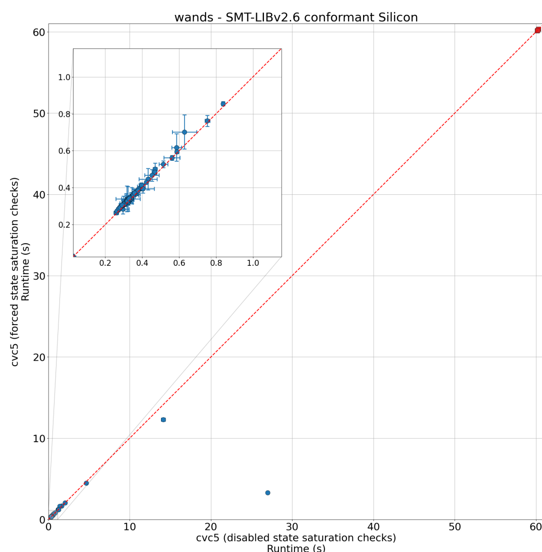

Figure A.33: Runtime comparison between SMT-LIBv2.6 Silicon using cvc5 with forced (y-axis) and disabled (x-axis) state saturation checks for the `wands` test directory. Data points are means $\pm$ standard deviation of 10 repetitions of every test in the standard Silicon test suite. Passed tests are blue circles and timed out tests are red squares. Failed, cancelled or ignored tests are not depicted.

Bibliography

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–30, 2019.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24. URL https://doi.org/10.1007/978-3-030-99524-9_24.

[3] 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 *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[4] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[6] 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. Available at www.SMT-LIB.org.

[7] Thomas Bouton, Caminha B de Oliveira, David Déharbe, Pascal Fontaine, et al. verit: an open, trustable and efficient smt-solver. In *International Conference on Automated Deduction*, pages 151–156. Springer, 2009.

[8] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL https://hal.inria.fr/hal-01960203.

[9] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[10] Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *International Conference on Computer Aided Verification*, pages 596–603. Springer, 2018.

[11] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[14] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014. doi: 10.3233/sat190101. URL https://doi.org/10.3233/sat190101.

[15] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Syntax-guided quantifier instantiation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163. Springer, 2021.

[16] Malte H Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

[17] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[18] 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)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SUPPORTING ALTERNATIVE SMT SOLVERS IN VIPER

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
ANTHONY	LASSE FREDERIK WOLFF

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
SINGAPORE, 28/04/2022	Lasse F. W. Anthony

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.