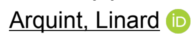
ETH zürich

ETH Library

Profiling Symbolic Execution

Master Thesis

Author(s):
Arquint, Linard 

Publication date:
2019

Permanent link:
https://doi.org/10.3929/ethz-b-000392350

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Profiling Symbolic Execution

Master's Thesis

Linard Arquint

arquintl@student.ethz.ch

Advisor

Dr. Malte Schwerhoff

Supervisor

Prof. Dr. Peter Müller

Programming Methodology

Institute for Programming Languages and Systems

Department of Computer Science

ETH Zürich

October 1, 2019

ETH zürich

Abstract

Symbolic execution [28] explores a program path-wise and queries a SAT modulo theory (SMT) solver with many formulae along the way. In comparison to a regular program execution, symbolic execution differs in several aspects: Symbolic instead of concrete values are used and all (feasible) program paths are explored. Already analyzing on which path most time was spent in a symbolic execution is a difficult question, especially without tool support.

This thesis presents a profiler for the symbolic execution engine Silicon [35]. The profiler consists of two parts, a logger to record a symbolic execution and an application to visualize the log. The log format is generic in the sense that it only makes basic assumptions about symbolic execution. Therefore, other symbolic execution engines should also be able to record their execution in the same log format and thus can reuse the visualization application. In addition, the logger can easily be extended to log additional steps of the symbolic execution.

Particular input programs for Silicon have been used to evaluate the profiler. The evaluations show that the profiler identifies similar performance culprits as manual inspection concludes. Furthermore, the profiler is a large timesaver reducing the time to analyze a symbolic execution from a couple of days, as needed for a manual inspection, to just a few hours.

Acknowledgements

I would like to express my special thanks to my advisor Dr. Malte Schwerhoff. He has guided me to the successful completion of my thesis and has brought up many ideas for additional features of the Visualizer. Additionally, the meetings with him have greatly helped me to see the big picture of the thesis and not getting lost or stuck in minor details.

I would also like to acknowledge Prof. Dr. Peter Müller for being my supervisor. I am very thankful for introducing me to this thesis' topic and giving me the opportunity of pursuing the Master's thesis in his group.

My gratitude also goes to my parents, Helen and Philipp, and girlfriend, Kristina, for their unfailing support and continuous encouragement throughput my years of study as well as while conducting research for this thesis.

Finally, thank you Jonas for enjoying the daily coffee together. The discussions with you were not only entertaining but have also provided me with fresh ideas when I needed them.

Zürich, October 2019

Linard Arquint

Contents

Abstract i

Acknowledgements iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Problem Statement . . . 2
  - 1.1.1 State of the Art . . . 2
  - 1.1.2 Motivation . . . 3
  - 1.1.3 Contributions . . . 4
- 1.2 Background . . . 4

2 SymbEx Logger 7
- 2.1 Previous Work . . . 8
- 2.2 The new SymbEx Logger . . . 10
  - 2.2.1 Logging in the Presence of Verification Failures . . . 13
  - 2.2.2 Log Output . . . 13
- 2.3 Continuous Integration . . . 14

3 Visualizer 17
- 3.1 Input Preprocessing . . . 18
- 3.2 Record Completion . . . 18
- 3.3 Duration Calculation . . . 20
- 3.4 Scope Creation . . . 23
- 3.5 Scopes . . . 25
  - 3.5.1 Scope Filtering . . . 25
- 3.6 Graphical Representation . . . 30
  - 3.6.1 JavaScript Console Integration . . . 35
- 3.7 Bottlenecks . . . 37
- 3.8 Continuous Integration . . . 38

4 Evaluation **39**
4.1 SymbEx Logger Evaluation 39
4.2 Visualizer Evaluation 42
4.2.1 Quantitative Evaluation 42
4.2.2 Qualitative Evaluation 43

5 Conclusions **53**

6 Future Work **55**

A Technical Details **57**
A.1 SymbEx Logger Configuration 57
A.2 SymbEx Logger Unit Tests 57

B Scope Filtering Analysis **61**

C Profiling Minutes **63**

Bibliography **79**

List of Figures

1.1 Overview of Viper's architecture showing the front-ends on the top and the two back-ends, one based on verification condition generation and the other, Silicon, on symbolic execution. This figure is taken from [32]. 5

2.1 Symbolic execution log for the Viper program in listing 2.2, showing the ID of each record on the left. The log's structure follows the symbolic execution, but nesting information is only implicitly available by interpreting open and close scope records. 11
2.2 Record types of the new SymbEx Logger 12

3.1 Pipeline for processing the exported symbolic execution log, generating the intermediate data structure, and visualizing it. The intermediate data structure can be exported as well as imported. 17
3.2 General structure of a scope. The characteristical records are shown and have corresponding timestamps t_{0-7}. To calculate the scope's duration, the individual path segments $a - e$ have to be considered. . . 21
3.3 A path for a particular scope is characterized by the open scope record, a close scope record and zero or more branch or join points in between. 21
3.4 Scope representation for the symbolic execution log in figure 2.1. Horizontal arrows denote the sub-scope, vertical arrows the successor, and dashed arrows the inter-level successor relation. 23
3.5 Sample of a scope data structure. Scope 1 is sub-scope of 0 and has two successors, namely scopes 2 and 3. Scope 2 has 5 as inter-level successor. Scope 3 is marked for removal. 26
3.6 Removal of scope 3 corresponding to case 1. 27
3.7 Removal of scope 3 corresponding to case 2. 27
3.8 Non-permitted removal of scope 3 in a special situation of case 2. . . . 27
3.9 Removal of scope 3 corresponding to case 3. 28
3.10 Removal of scope 3 corresponding to case 4. 28
3.11 Scope data structure needing two filtering passes. 29
3.12 Visualization of the symbolic execution of a small Viper program. . . . 31

3.13 Selection of two path segments, drawn with a pink border, in the Visualizer. 32
3.14 Histogram of the execution time for the selected path segments in the Visualizer. 32
3.15 Flamegraph of the execution time for the selected path segments in the Visualizer. 33
3.16 Detail of the Trace-Viewer [17] showing branching of a symbolic execution. 34
3.17 Settings popup of the visualizer offering various configuration possibilities. 35

4.1 Average execution time of each pipeline step based on symbolic execution logs of 16 Viper programs. 43
4.2 Profiling minutes for analyzing the symbolic execution of the Viper program composite.vpr 45
4.3 Page 1 of the profiling minutes for the symbolic execution of the Viper program borrow_first.rs.vpr 47
4.4 Page 2 of the profiling minutes for the symbolic execution of the Viper program borrow_first.rs.vpr 48
4.5 Page 1 of the profiling minutes for the symbolic execution of the Viper program Knuth_shuffle.rs.vpr 49
4.6 Page 2 of the profiling minutes for the symbolic execution of the Viper program Knuth_shuffle.rs.vpr 50
4.7 Profiling minutes for the symbolic execution of the Viper program RSLSpinlock.sil 51

B.1 Scope data structure that does not permit the removal of scope 3. . . . 61

C.1 Profiling minutes for analyzing the symbolic execution of the Viper program Slow.sil 64
C.2 Profiling minutes for a second analysis of Slow.sil including a direct comparison to a very similar but faster variant of the program. 65
C.3 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program RelAcqDblMsgPassSplit.sil 66
C.4 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program RelAcqDblMsgPassSplit.sil 67
C.5 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program AVLTree.nokeys.sil 68
C.6 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program AVLTree.nokeys.sil 69
C.7 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program 100_doors_generic.rs.vpr 70
C.8 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program 100_doors_generic.rs.vpr 71

C.9 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Ackermann_function.rs.vpr 72
C.10 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Ackermann_function.rs.vpr 73
C.11 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Fibonacci_sequence.rs.vpr 74
C.12 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Fibonacci_sequence.rs.vpr 75
C.13 Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Towers_of_Hanoi_spec.rs.vpr 76
C.14 Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Towers_of_Hanoi_spec.rs.vpr 77

1
Introduction

The Software Fail Watch [38] estimates that software failures in 2017 have resulted in costs of more than 1.7 trillion USD world-wide. Therefore, increasing the reliability of software is of interest globally and across industries. Testing is today the standard method for checking program correctness. However, testing only allows checking correctness for finitely-many potential program executions by providing particular input values and checking the results against expected output.

In contrast, program verification reasons about a program taking all its potential executions into account. It tries to find a proof for the absence of failures, for example that an assertion cannot be violated. Hence, program verification has gained interest, also thanks to advances in the performance of SAT modulo theory (SMT) solvers. A formula can be given to an SMT solver, which can either prove its satisfiability, give a counterexample, which violates it, or respond with unknown. Thus, an SMT solver is an important component of most program verification frameworks, because it is used to discharge proof obligations.

Symbolic execution. This thesis focuses on verification based on symbolic execution [28]. Symbolic execution is a program analysis technique following the simple principle of analyzing a program by breaking it up into many steps, often corresponding to individual statements and expressions. Besides for verification, it was proposed in the past for several different applications, such as test case generation [15, 24] or taint analysis [23, 34]. During symbolic execution, a stepwise execution of an input program is performed and symbolic instead of concrete values are used. For each step, a matching rule is selected from a set of symbolic execution rules. Each rule describes the effect of executing a step on the symbolic execution state. Therefore, at each program point, the symbolic execution

state captures the constraints on the symbolic values resulting from executing the input program up to this point.

Symbolic execution in the context of program verification uses the collected constraints on symbolic values to construct formulae for an SMT solver. It queries the SMT solver with many (and ideally comparably simple) formulae while exploring the program path-wise. Hence, symbolic execution typically queries the SMT solver often, but with relatively simple formulae, in contrast to, for example, tools based on weakest precondition calculi.

In the context of this thesis, the entire verification software using symbolic execution including the SMT solver is denoted by *symbolic execution framework*. In contrast, the *symbolic execution engine* is the entire symbolic execution framework except the SMT solver.

Profiling. The second part of this thesis' title is profiling. Profiling is the act of analyzing a program to find potential performance bottlenecks, such as memory consumption or execution time, with the goal of optimizing them. A profiler is typically used to gather the required information at runtime, i.e. while the program is being executed. Processing the gathered data and visualizing them to the user can either happen simultaneously with the program execution or afterwards, which is known as online and offline profiling, respectively. The visualization should be understandable and goal-oriented: a user has to be able to map each element in the visualization to some program part and understand why it exists and how it is influenced by the program. The visualization should provide insights that enable users to optimize their programs.

As a result, profiling symbolic execution is concerned with finding performance bottlenecks in a symbolic execution of a particular input program. Symbolically executing an input program causes many additional operations that are not directly visible in the input program's text, for example manipulating the symbolic state. Therefore, potential bottlenecks, identified by profiling, can reside in the input program, in the symbolic execution framework, or be the result of their combination. Nevertheless, identifying potential bottlenecks is the first step in resolving performance problems and speeding up the process of program verification.

1.1 Problem Statement

1.1.1 State of the Art

Symbolic execution is employed for various applications, reaching from test case generation [15,24] over block chain analysis [23] to program verification [35]. Tool support for debugging a symbolic execution has already been proposed in the past: Aurecchia [4] has explored debugging that focuses on visualizing symbolic execution states and providing a visual representation of a counterexample if the

program verification fails. Hentschel et al. [27] visualize in their symbolic execution debugger SED the reachable program states of the input program. SED uses KeY [7] as underlying symbolic execution engine, which in addition is capable of showing all symbolic execution steps performed during the proof construction. SED and the debugger by Aurecchia both aim to speed up the process of finding defects in an input program. Bornholt and Torlak [8] have presented the symbolic profiler SymbPro. Their work focuses on ranking locations of potential bottlenecks and visualizing them as a flamegraph. However, the profiler presented in this thesis centers on a visualization of the entire symbolic execution and tries to steer the user's attention to steps of the symbolic execution that take longer than others.

1.1.2 Motivation

Several symbolic execution frameworks, such as Rosette [37], EXE [15], its successor KLEE [14], or Silicon [35], have been proposed in the past. Although similarities exist, for example that they explore paths, neither standard approaches nor common tools for debugging or profiling exist. However, tool support is crucial for analyzing the execution time of a symbolic execution and finding performance culprits. From the perspective of an author verifying his program, a symbolic execution framework is a black-box taking a certain amount of time to symbolically execute the program. The total execution time is composed of the time spent in the symbolic execution engine as well as in the underlying SMT solver. Several papers [5, 14–16] have reported that the SMT solver significantly influences the verification's execution time. In addition, several optimization techniques have been proposed trying to speed up SMT queries, for example by caching or shrinking them [14, 15]. However, the symbolic execution engine itself performs potentially costly operations, for example managing the symbolic state. Furthermore, the time spent in the SMT solver directly results from the SMT queries that the symbolic execution engine repeatedly issues.

Therefore, we see the potential for a symbolic execution profiler that is capable of visualizing the individual steps that a symbolic execution engine performs. The profiler should enable more users to understand the operations of a symbolic execution engine and the verification's execution time of a particular input program by not requiring expert knowledge about the used framework. Furthermore, these insights can be used to optimize the input program or the symbolic execution engine. As reported, a good starting point for a profiler is to determine the split between work done by the SMT solver and the symbolic execution engine. Already simple information, such as the total execution duration of each part, could be used to evaluate whether a particular symbolic execution was significantly influenced by the SMT solver. Additionally, the same information could be used to monitor the performance gains for a potential optimization.

1.1.3 Contributions

Logger. First of all, profiling requires extensive data about a program execution. Chapter 2 starts with an illustration why the existing logging infrastructure of Silicon is not able to collect the required data. Hence, it motivates the creation of a new logging infrastructure, the first major contribution of this thesis. It records not only the duration of individual symbolic execution steps but also their relations between each other, to correctly represent the symbolic execution and to enable a visualization of the symbolic execution that is understandable and goal-oriented. Also part of the contribution is a logging data format that allows to store the collected data and to later visualize them.

Visualizer. As a second contribution, this thesis presents a visualization pipeline (chapter 3) that processes the collected symbolic execution log to generate an intermediate data representation. Based on the intermediate data representation, the program paths and, additionally, metrics of the symbolic execution are graphically displayed to the user. Furthermore, an application programming interface (API) is provided to efficiently perform queries on the data. The API is especially useful to prototype and evaluate additional visualization features before actually implementing them with a proper user interface (UI).

Implementation. As a third contribution, this thesis adds an implementation of the new logging infrastructure to Silicon, replacing the old one, as well as an implementation of the visualization pipeline as a JavaScript application for all common web browsers.

1.2 Background

Although most concepts presented in this thesis apply to other verifiers based on symbolic execution, the focus lies on Silicon [35].

Silicon is one of two back-ends of the Viper infrastructure [32] and is capable of verifying input programs written in the Viper language. Figure 1.1 gives an overview over the Viper infrastructure. The second back-end is based on verification condition (VC) generation and compiles a Viper program to a Boogie program and forwards it to Boogie [30]. Several front-ends compile different source languages to the intermediate language Viper, which supports specifications. Hence, the intermediate language is agnostic to the source language. Furthermore, it facilitates verification by providing reasoning about permissions and having controlled heap access based on permissions.

Silicon performs a modular program verification by verifying each procedure and separation logic predicate [33] individually. Furthermore, the program, respectively its control flow graph (CFG), is explored path-wise. Whenever branching occurs, for example due to if-else statements or ternary operators, Silicon branches

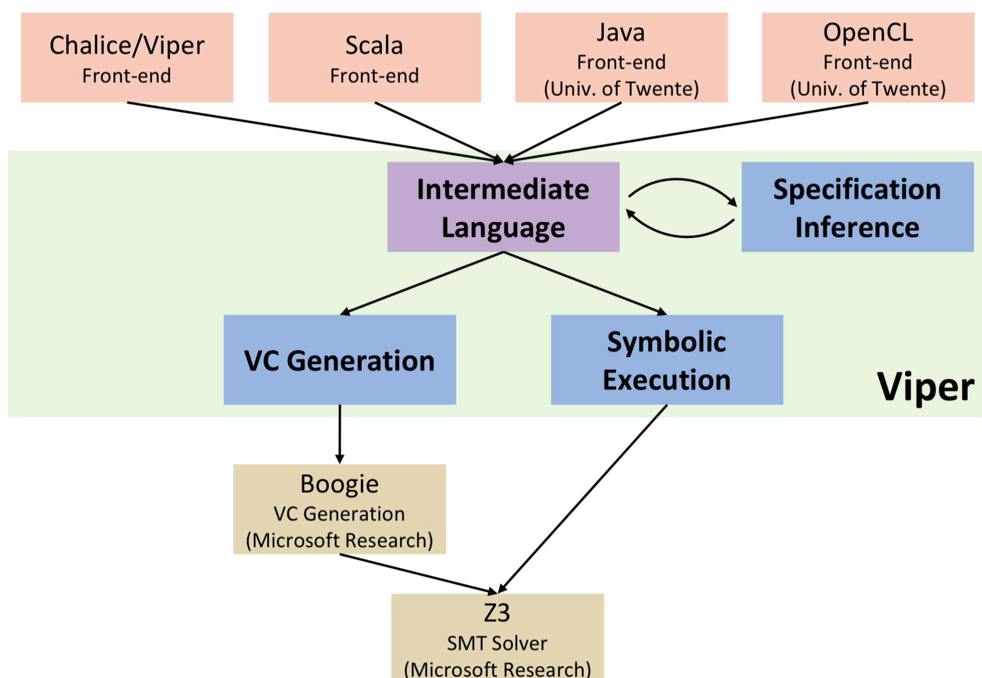


Figure 1.1: Overview of Viper's architecture showing the front-ends on the top and the two back-ends, one based on verification condition generation and the other, Silicon, on symbolic execution. This figure is taken from [32].

the execution and individually explores each path. Silicon repeatedly queries the underlying SMT solver along the paths, for example to check whether a branch is feasible or an assertion can be violated. As underlying SMT solver, Silicon uses Microsoft Research's Z3 [20].

Assuming Silicon performs only branching, the explored paths form a tree structure. However, Silicon joins the execution if two branches have semantically equivalent heaps and thus differ only in their path conditions and symbolic expressions. This situation occurs if the evaluation of an expression results in branching, because expressions are pure, i.e. do not modify the heap. Joining reduces the number of paths explored, but increases the complexity of path conditions, because the path conditions of each branch have to be combined by conditionals. In presence of branching and joining, the resulting symbolic execution, i.e. the explored paths, can no longer be represented as a tree, but form a directed acyclic graph (DAG) instead.

In the remainder of this thesis, chapter 2 shows the limitations of the old symbolic execution logger and motivates the creation of a new one. The Visualizer, which takes a symbolic execution log as input and provides a UI that enables users to analyze the log, is described in chapter 3. We evaluate the profiler in chapter 4 and conclude (chapter 5) that it is a large timesaver in comparison to manually

inspecting a symbolic execution. Chapter 6 presents some topics that could be explored in future work.

2

SymbEx Logger

Gathering information about a program execution is crucial for profiling. For profiling a symbolic execution of a particular input program, the steps performed by the symbolic execution framework need to be recorded, because it is a priori unknown which steps are relevant. Logging a symbolic execution has two major properties:

Firstly, records might not only follow one by one, but can be nested. This information about nesting should be captured in the log in order that steps can be distinguished from sub-steps. Earlier work by Colombo [18] identified an adjustable level of detail as a main design principle:

> The user should always have the choice to select the level of detail of the information that is displayed. The amount of data is too large to be comprehended otherwise.

Having hierarchical information in the collected log, a UI can use it to allow users to show or hide details of a step and therefore adjusting the level of detail. Thus, a user can quickly find an important top-level step and zoom in by revealing its details. This first property is not unique to symbolic execution, but also applies to a regular execution of a program, because, for example, statements also have sub-steps, i.e. subexpressions.

Secondly, symbolic execution might branch as well as join and execute several branches. Hence, a meaningful log needs to remember which branch logged which records. The resulting log is therefore non-linear, in contrast to a regular execution of a program.

This thesis explores offline profiling, meaning that complete information about an execution is collected first, before analyzing it.

Listing 2.1: Viper program with an if-else statement

```
method m1(i: Int) returns (res: Int)
{
  var j: Int
  j := 2
  if (i < j) {                      // 5
    j := 3
  } else {
    j := 4
  }
  res := j                          // 10
}
```

2.1 Previous Work

Silicon already has a logging infrastructure called SymbEx Logger [13]. During the symbolic execution, it keeps track of inserted records and tries to correctly capture sub-records. The implementation has both aforementioned properties of symbolic execution logs.

However due to the complexity of the logger's implementation, refactoring or extending Silicon often resulted in breaking the logger. When we have started working on this thesis, the logger was unable to correctly track branching. Furthermore, it required various hacks to support branching that lead, in certain situations, to wrong results.

Listing 2.1 shows a simple Viper program to illustrate the logging mechanism. Silicon branches at line 5, because both branches (`i < j` and `i >= j`) are feasible. For if-else statements, Silicon does not join the symbolic execution. Therefore, `res := j` will be execute twice, once on each branch. To correctly represent this in the log, a branching record will be inserted into the log. A branching record keeps track of the records for each of its two branches. For this particular example, the scope of the two branches extends to the end of the method, because no joining occurs. The records for the assignments `j := 3` resp. `j := 4` and `res := j` will occur on each branch.

Listing 2.2 shows a Viper program, for which the old SymbEx Logger does not correctly record the symbolic execution. The corresponding log is shown in listing 2.3, where an increased level of indentation indicates that a record is a sub-step. `inhale b ? acc(x.f) : acc(x.g)` adds a permission to the field `f` resp. `g` of object `x` depending on condition `b`. Due to the fact that permissions are involved in this ternary operator, this is impure branching and Silicon will not join the two branches before executing line 7. In the log, a branching record is located on line 4 as sub-step of the `inhale` execution. Furthermore, the evaluation of condition `b` and of `acc(x.f)` as well as `acc(x.g)` are correctly logged.

Listing 2.2: Viper program with an impure conditional expression.

```
field f: Int
field g: Int

method m2(b: Bool) returns (res: Int)
{
  inhale b ? acc(x.f) : acc(x.g)
  res := 1
}
```

Listing 2.3: Symbolic execution log of the old SymbEx Logger for the Viper program in listing 2.2. An increased level of indentation indicates that a record is a sub-step. "Condition", "Branch 1", and "Branch 2" were inserted to better distinguish their sub-steps.

```
method m2
  exec inhale b ? acc(x.f) : acc(x.g)
    eval b ? acc(x.f) : acc(x.g)
      Branching
        Condition
          eval b
        Branch 1
          eval acc(x.f)
          exec res := 1
            ...
        Branch 2
          eval  acc(x.g)
          exec res := 1
            ...
```

However, a record representing the execution of `res := 1` is added to each branch as well (on lines 9 and 13). Hence, it looks like as if `res := 1` is also a sub-step of the **`inhale`** execution. This is incorrect and results from the fact that `res := 1` has to be placed on each branch and both branches are part of the branching record, which in turn is a sub-step of the **`inhale`** execution.

Correctly reconstructing the symbolic execution based on this (faulty) log is non-trivial. Additional reasoning or metadata would be necessary to capture the fact that the assignment is actually a statement following the **`inhale`** statement instead of being a subexpression of the inhale statement.

Instead of performing these log transformations to correctly reproduce the symbolic execution, we decided to change the logger to generate a log that correctly reflects the symbolic execution in the first place. We initially assumed that the existing logger is capable enough of collecting the necessary data about a symbolic

execution, thus replacing it with a new logger was unexpected.

2.2 The new SymbEx Logger

The key idea of the new SymbEx Logger is that almost every operation during the symbolic execution is a scope – it has a fixed start and end point in time, it contains some payload (e.g. the statement that it represents), and it might contain sub-scopes. The only exception are branching records: They are still inserted into the log whenever the symbolic execution branches. The main problem of the old SymbEx Logger is, that branches are considered scopes as well. Therefore, branches have to respect proper nesting and hence have to end before the parent scope ends. Specifically for listing 2.3, this means that the two branches cannot continue after the **`inhale`** statement, because they are sub-scopes of it. As a consequence, the old SymbEx Logger has no other choice than adding the records for `res := 1` as sub-scopes of the **`inhale`** statement.

By not treating branching records and branches as scopes, they do not have to maintain proper nesting and can continue. This treatment might appear artificial at first. However, it becomes a natural concept when thinking of a branching record as a single point in time, i.e. the point where the symbolic execution branches.

Figure 2.1 shows the log of the new SymbEx Logger for the symbolic execution of the same Viper program (from listing 2.2). Records starting with the prefix "Data" are data records and store the payload of a symbolic execution operation. They correspond to the records of the old SymbEx Logger. Directly following each data record there always is a single open scope record. It references the data record and marks the beginning of the scope. The scope continues until it is eventually ended by a close scope record referencing the same data record. If branching occurs in the scope, there might be multiple close scope records for this specific record. For example, the data record with ID 2, representing the execution of the **`inhale`** statement, is closed on the first branch by record 14 and on the second branch by record 22. However, there has to be exactly one close scope record for each open scope record on each path.

The key difference to the old SymbEx Logger emerges as well in figure 2.1: Due to the fact that branching is not treated as a scope, the branches can continue independently of scopes. Consider for example the first branch: The **`inhale`** statement's scope 2 is closed by record 14 and scope 15 of the `res := 1` statement starts at record 16. The branch itself starts somewhere during the execution of scope 2 but continues and outlasts scope 15.

Figure 2.2 shows the three basic types of records in the new logger. As already seen in the example log, a data record describes a symbolic execution operation. Beside a label, data records have additional fields to keep track of further information, for example whether it represents an SMT query or whether the sub-scope

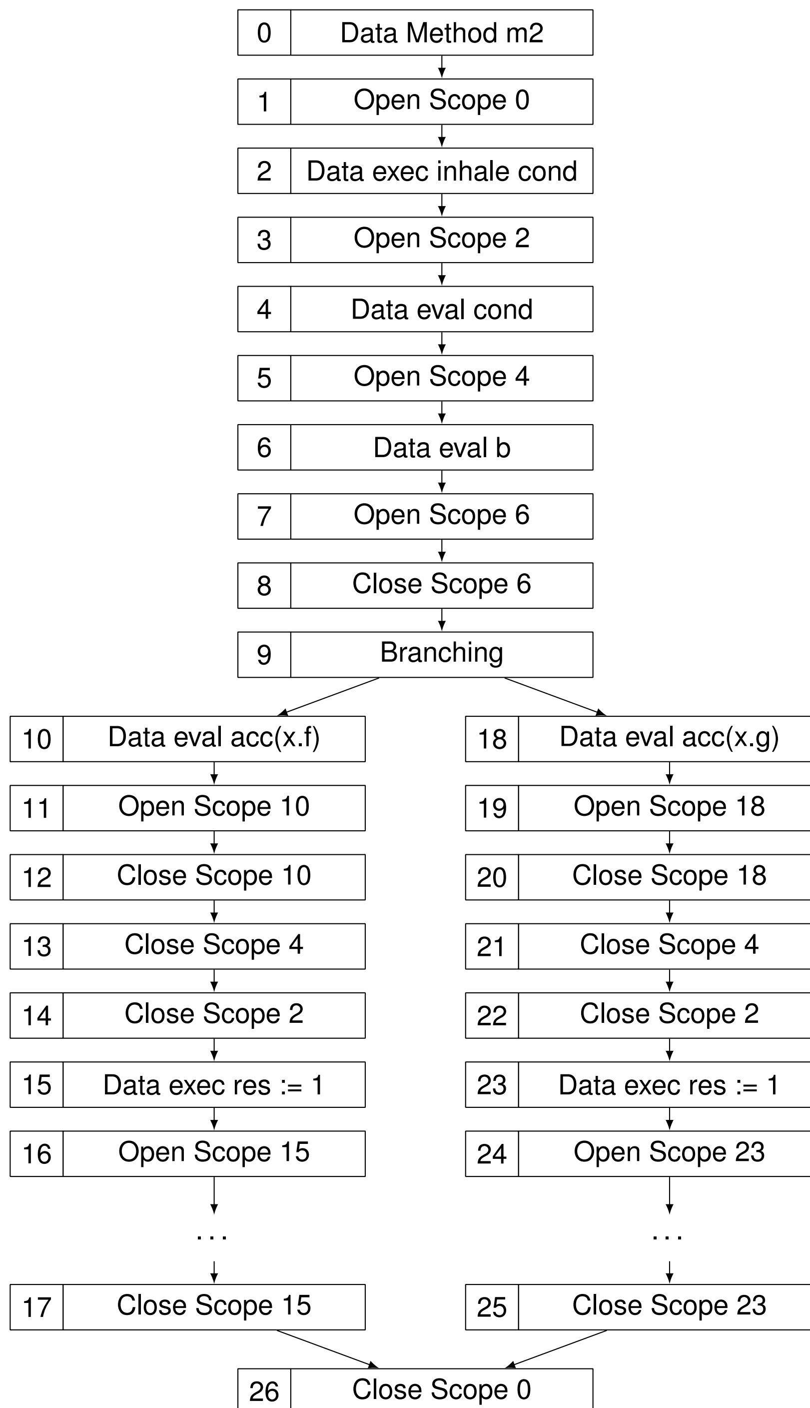


Figure 2.1: Symbolic execution log for the Viper program in listing 2.2, showing the ID of each record on the left. The log's structure follows the symbolic execution, but nesting information is only implicitly available by interpreting open and close scope records. ”cond” is used as an abbreviation for `b ? acc(x.f) : acc(x.g)`.

Data Record	Open / Close Scope Record	Branching Record
• Label • Is SMT query	• Reference data record • Time	Per branch: • Is reachable • Start time • List of records

Figure 2.2: Record types of the new SymbEx Logger

contains joining. Having such information explicitly available in the log simplifies a later analysis, because an analysis does not need to make assumptions based on the label (i.e. whether a record represents an SMT query) or to detect joins in the sub-scopes.

Open and close scope records do not only act as markers for the beginning and end of a scope, but they also provide the start respectively end time. Furthermore, they reference the data record to which they belong. This reference exists for convenience reasons and can be used for checking consistency. However, each scope has to either be fully contained in another scope or not at all, meaning that a scope cannot start in one parent scope and end in a different one. Therefore, close scope records can unambiguously be matched with open scope records.

Branching records do not simply keep track of the individual branches, but provide additional information too. For each branch, a flag indicates whether it is reachable or not. In addition, the start time of each branch is stored as well. As it will be shown in section 3.3, each branch's start time is necessary to correctly calculate the duration of a scope.

The new logger not only produces a log output that more closely resembles the steps that the symbolic execution framework has performed but can also be configured to log only a certain subset of records (see section A.1 in the appendix). In addition, the new logger heavily simplifies its implementation, because the burden of keeping track of the individual scopes is completely removed from the logger by just inserting open and close scope records into the log. Therefore, calls to the logger infrastructure can almost directly be converted to corresponding log insertions. Furthermore, the logger does not need to keep a stack of currently open scopes as was previously necessary. The logger just needs to keep track of all branching records, so that records are added to the currently active branch.

The logger's simplification should result in better maintainability for upcoming changes in Silicon. Section 2.3 presents additional measures that were taken to ease maintenance. Although the new logger was simplified compared to the old one, the difficulty of deciding which record is a sub-scope of which other record is not just gone. As chapter 3 will show, several postprocessing steps are performed on a symbolic execution log. One of them uses the open and close scope records to make the sub-scope relation explicit. Hence, some work that

the old SymbEx Logger performed was shifted to a postprocessing step.

2.2.1 Logging in the Presence of Verification Failures

The discussion so far has assumed that the verification of the input program succeeds. Under this assumption, all scopes will properly be closed in the resulting log. However, if the verification fails, Silicon quits its symbolic execution and reports the verification failure. Therefore, some scopes will not properly be closed, meaning that an open scope record will be present but the corresponding close scope is missing on some paths. Instead of completing the log, the specifications for log files have been weakened to allow incomplete logs. Thus, scope completion can be deferred to the post-processing of the log (see section 3.2), keeping the logger itself simple.

2.2.2 Log Output

As soon as Silicon finishes the verification, the collected log is serialized. Firstly, all references to other records are replaced by their IDs. Although there is a proposal for supporting references to objects in JavaScript object notation (JSON) (RFC 6901 [12]), it is not yet in wide use. Therefore, we convert references to IDs making the resulting serialization fully compliant with the latest JSON format standard (RFC 8259 [10]). Next, all records are serialized to a flat JSON array, with each entry conforming to the interface in listing 2.4. The serialized log is reported by Silicon to interested clients. This report feature is used for integrating the new SymbEx Logger into ViperServer, a HyperText transfer protocol (HTTP) server interface for Viper. In particular, ViperServer and the logger's integration enable clients to be decoupled from Silicon, simply submit verification tasks, and still have access to the resulting symbolic execution log. Furthermore, a client can decide on its own what to do next with the received log. Besides writing it to disk, a client might process the log and generate visualizations. The current implementation reports the entire log as a single execution trace report and uses spray-json [36], a JSON implementation in Scala. Serializing the entire report at once has the disadvantage that its serialization has to fit into memory, which is not the case for large reports. This issue could be mitigated by reporting partitions of the entire log and thus serializing only a fraction of the entire log at a time.

Note that the serialization does not distinguish between data, open and close scope, or branching records. Instead, individual fields indicate the record's original role. For example, lines 6 and 7 in listing 2.4 indicate whether a record is an open or close scope record. A record corresponding to a branch point stores an array of `BranchInfo` (line 9). Each array entry corresponds to a single branch and stores whether the branch is reachable (line 14). Furthermore, the start time, at which the symbolic execution has started exploring the branch, is stored as well (line 15). Line 16 lists the IDs of the records that lie on the branch.

Listing 2.4: Interface for each log record's serialization

```
export default interface Record {
  id: number;
  kind: string;
  value: string;
  isJoinPoint?: boolean;
  isScopeOpen?: boolean;
  isScopeClose?: boolean;
  isSyntactic?: boolean;
  branches?: BranchInfo[];
  data?: Data;
}

export interface BranchInfo {
  isReachable: boolean;
  startTimeMs: number;
  records: number[];
}

export interface Data {
  refId?: number;
  isSmtQuery?: boolean;
  timeMs?: number;
  [key: string]: any;
}
```

Information that does not describe the symbolic execution's structure is separately placed in `data` (line 10). Open and close scope records store there the ID of the data record to which they belong (line 20). Furthermore, line 23 allows additional key-value pairs being present in the data object. This feature is currently used for attaching statistical information of the SMT solver to individual records, but additional data can be attached as well.

Various field names end in a question mark indicating that these fields are optional. This is a simple optimization for reducing the log's size, because most records require only a subset of all fields.

2.3 Continuous Integration

Although the SymbEx Logger implementation was heavily simplified in this work, keeping the logger up-to-date with Silicon is crucial. For this reason, unit tests check, for certain input programs, whether the obtained logs correspond to the expected logs. This functionality was already present in the old SymbEx Logger but was now significantly improved. Checking that an obtained log is identical to an expected log leads to many false positives when only small changes or

optimizations are performed in Silicon, or if Silicon is extended to log additional symbolic execution operations. Therefore, the unit tests were adapted to use the expected logs just as a minimal log, meaning that an obtained log can contain more records, but has to have at least the records from the expected log and preserve the sub-scope relationship. More details about the comparison of the log output can be found in the appendix (section A.2).

In addition, the expected logs provided to the unit tests do not simply correspond to the current output of the SymbEx Logger. The expected logs only contain records representing members (functions, methods, and predicates), statements, as well as structural records (branching and joining). We believe that this is a good tradeoff between being flexible enough for upcoming changes and strict enough to detect regressions.

3
Visualizer

The Visualizer is a TypeScript application taking the log of a symbolic execution as input and graphically displays it. In addition, it allows users to interact with it. The application can be opened and run in any common web browser.

Figure 3.1 shows the entire pipeline from reading the input file containing the symbolic execution log, up to graphically visualizing the log. The intermediate data structure, created in step 4, can be exported as a JSON file. When importing a JSON file containing the intermediate data structure, the first four pipeline steps will be skipped.

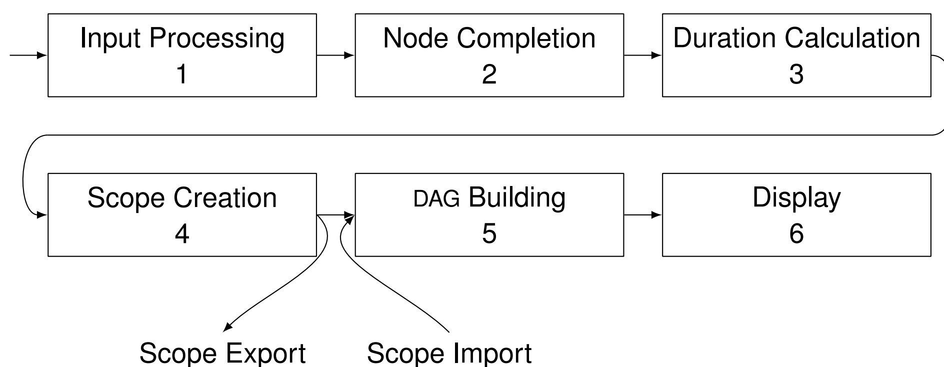


Figure 3.1: Pipeline for processing the exported symbolic execution log, generating the intermediate data structure, and visualizing it. The intermediate data structure can be exported as well as imported. Not shown are several transformations that are performed on the intermediate data structure immediately before visualizing it. These transformations include filtering syntactic nodes and highlighting the duration-wise longest path.

This chapter presents certain important steps from the pipeline and ends with features provided by the graphical visualization.

3.1 Input Preprocessing

The Visualizer utilizes the web browser's file selection dialog in order that users can select a JSON file containing the symbolic execution log of a previous verification. This allows easy repetition of profiling a certain verification and it decouples profiling from the verification toolchain. Because the log format is not specially tailored to the symbolic execution by Silicon, other symbolic execution frameworks should also be able to record logs the same way. Therefore, the Visualizer is agnostic to the symbolic execution framework creating the log.

Besides loading the log file, the second part of the input preprocessing is replacing ID references by an actual reference to the referenced record. As seen in the previous chapter, each record stores the ID of the next record(s) in the log file, because the JSON standard does not allow references. After preprocessing the input, each record stores a reference to its successor(s).

3.2 Record Completion

As mentioned in the previous chapter, the symbolic execution log can be incomplete, especially when a verification failure occurs. An incomplete log is characterized by missing close scope records. The algorithm shown in listing 3.1 locates scopes that are not correctly closed and inserts corresponding close scope records.

The algorithm takes a preprocessed root record as input and assumes that at least the root's scope is complete. This assumption holds for the current Silicon implementation, because member records, including their open and close scope records, are handled differently compared to regular data records. The algorithm iterates over all records including the newly created records in topological sort order (line 2). Line 1 creates `stackMap`, a mapping from records to scope stacks. The function `complete` is invoked for each scope. First, the scope stack is retrieved from the mapping (on line 5). The scope stack corresponds to the scopes that are open before taking the current record into account. If the record is an open scope record then the scope is pushed onto the stack. In contrast, if it is a close scope record, the stack is popped, because the inner-most (i.e. the stack's top) scope was just closed. Lines 11 and 12 check for each direct successor, whether it respects the scope stack or a close scope record is missing. Finally on lines 14 and 15, the current stack is duplicated and set as initial stack for each direct successor.

The function starting on line 17 takes a direct successor as well as the current stack and returns a (potentially different) successor. Line 18 first checks whether

Listing 3.1: The record completion algorithm.

```
let stackMap := [root.id -> []];
topSort(root).forEach(complete);

func complete(record) do
  let stack := stackMap[record.id];
  if record.isScopeOpen then
    stack.push(record.refId);
  else if record.isScopeClose then
    stack.pop();

  record.successors := record.successors
    .map(insertSuccessors(stack));

  record.successors.forEach((successor) =>
    stackMap[successor.id] = stack.copy());

func insertSuccessors(stack, successor) do
  if !successor.isScopeClose then
    return successor;
  let stackCopy := stack.copy();
  let expectedRefId = stackCopy.pop();
  let newRecords := [];
  while expectedRefId != successor.refId do
    newRecords.add(new ScopeCloseRecord());
    expectedRefId := stackCopy.pop();
  linkRecords(newRecords, successor);
  if newRecords.isEmpty then
    return successor;
  return newRecords[0];
```

the successor is a close scope record. If it is not, the same successor is simply returned. The algorithm only checks for close scope records whether they conform to the scope stack. On line 20, the current stack is duplicated because there might be multiple successors and each successor should start with the same stack. The inner-most scope ID is retrieved on line 21. In case the successor closes a different scope than expected by the stack, a close scope record is missing and is created on line 24. There might be multiple missing close scope records. Thus, the `while` loop adds them until the successor corresponds to the closing of the inner-most scope. Line 26 invokes another function that sets the successor relation for each newly created close scope record, such that each new record has the next new record as successor respectively the last new record has `successor` as successor. The function either returns the original successor, if no new records have been created, or the first created record (lines 27–29).

Each newly inserted close scope record gets a unique ID, a reference to the data node (the stack's top at the time of creation), and a timestamp marking the scope's end time. Terminating the symbolic execution in case of a verification failure is assumed to take zero time. Hence, the new close scope records get the same timestamp as the next close scope record that was already present (i.e. `successor`). Based on the algorithm's initial assumption that the outer-most scope is complete, there is not only always a next close scope record but it also is sufficient to only check for missing close scope records when hitting a close scope record.

3.3 Duration Calculation

Thanks to the previous step, the duration calculation step can assume a complete log. Using the timestamps from open and close scope records and calculating a duration is more involved than it looks like at the first glance. In particular, the following factors have to be considered, all illustrated in figure 3.2: Firstly, a scope can have multiple close scope records, as represented by the timestamps t_6 and t_7 in the figure. Considering the paths between the open scope and all close scope records of a specific scope, some path segments might be shared and therefore should only be counted one. For example, segments a and c are shared by all paths. Next, a branching record stores for each branch a time-stamp at which the exploration started. These timestamps have to be used as starting times for the individual branches, because the symbolic execution might have spent time in other scopes after a close scope record. In figure 3.2, the branches could continue after the close scope records and the duration of the second branch should therefore be calculated as $t_7 - t_5$. Last but not least, the scope might include joining. Considering the same figure, the duration of segment b should be computed as $t_3 - \min\{t_1, t_2\}$, because the symbolic execution has sequentially explored both branches when reaching the join point at t_3.

The implemented algorithm first collects all data records. It then individually calculates the duration for each data record respectively the corresponding scope. For a selected scope s, the algorithm iterates backwards over the data structure to collect all records characterizing the duration of s per path. As shown in figure 3.3, such a path ends in a close scope record ending scope s. Its start is marked by the (unique) open scope record of s. Between them, there can be zero or more branch and join points, each specifying an end and start time for the previous respectively next path segment. After a pass over all records, the algorithm will have collected the records characterizing the durations of all paths.

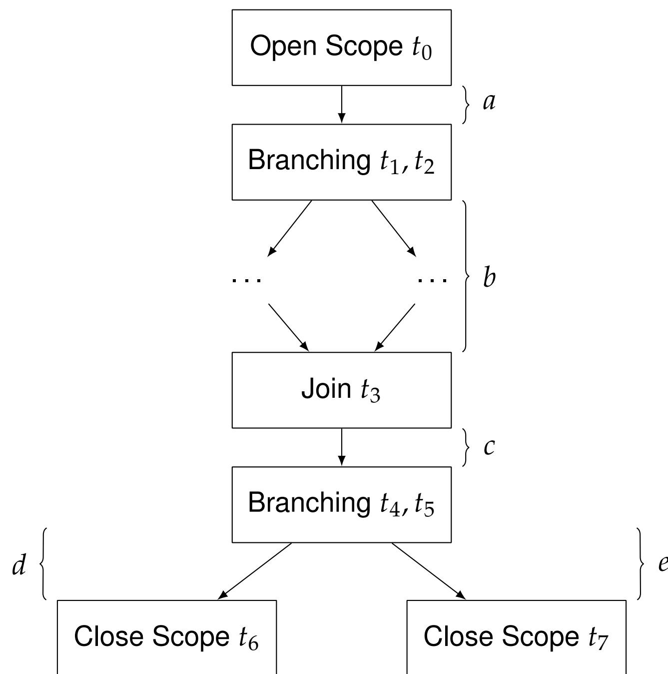


Figure 3.2: General structure of a scope. The characteristical records are shown and have corresponding timestamps t_{0-7}. To calculate the scope's duration, the individual path segments $a - e$ have to be considered.

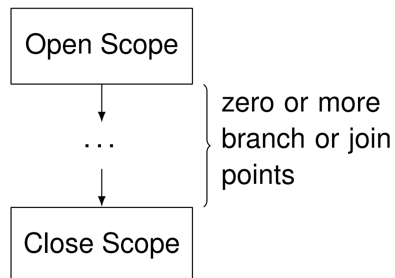


Figure 3.3: A path for a particular scope is characterized by the open scope record, a close scope record and zero or more branch or join points in between.

$$
\begin{aligned}
&(t_0, \min\{t_1, t_2\}), (\min\{t_1, t_2\}, t_3), (t_3, \min\{t_4, t_5\}), (t_4, t_6) \\
&(t_0, \min\{t_1, t_2\}), (\min\{t_1, t_2\}, t_3), (t_3, \min\{t_4, t_5\}), (t_5, t_7) \\
&(t_0, \min\{t_1, t_2\}), (\min\{t_1, t_2\}, t_3), (t_3, \min\{t_4, t_5\}), (t_4, t_6) \\
&(t_0, \min\{t_1, t_2\}), (\min\{t_1, t_2\}, t_3), (t_3, \min\{t_4, t_5\}), (t_5, t_7)
\end{aligned}
\tag{3.1}
$$

$$
\begin{aligned}
&(t_0, \min\{t_1, t_2\}), (\min\{t_1, t_2\}, t_3), (t_3, \min\{t_4, t_5\}), (t_4, t_6) \\
&\cancel{(t_0, \min\{t_1, t_2\})}, \cancel{(\min\{t_1, t_2\}, t_3)}, \cancel{(t_3, \min\{t_4, t_5\})}, (t_5, t_7) \\
&\cancel{(t_0, \min\{t_1, t_2\})}, \cancel{(\min\{t_1, t_2\}, t_3)}, \cancel{(t_3, \min\{t_4, t_5\})}, \cancel{(t_4, t_6)} \\
&\cancel{(t_0, \min\{t_1, t_2\})}, \cancel{(\min\{t_1, t_2\}, t_3)}, \cancel{(t_3, \min\{t_4, t_5\})}, \cancel{(t_5, t_7)}
\end{aligned}
\tag{3.2}
$$

$$
\begin{aligned}
&\underbrace{(\min\{t_1, t_2\} - t_0)}_{a} + \underbrace{(t_3 - \min\{t_1, t_2\})}_{b} + \underbrace{(\min\{t_4, t_5\} - t_3)}_{c} \\
&\qquad + \underbrace{(t_6 - t_4)}_{d} + \underbrace{(t_7 - t_5)}_{e}
\end{aligned}
\tag{3.3}
$$

Equation 3.1 shows the four paths in figure 3.2 expressed in terms of their segments. Each segment is a tuple of the segment's start and end, which are declared as the corresponding record (omitted in the equation) and its timestamp. As mentioned in the introduction of this section, simply calculating the duration per path and summing them up results in a wrong duration for scope s. Therefore, the algorithm performs a merge step that identifies and removes duplicate path segments. The merge iterates over all segments of all paths and removes segments with the same start and end record and timestamp. Equation 3.2 shows the remaining path segments after merging. Based on the unique path segments, the scope's duration can be calculated by subtracting each segment's start from its end timestamp and adding them, as seen in equation 3.3. $a - e$ refer to the path segments as annotated in figure 3.2.

Iterating over the entire data structure (once) for each data record as well as keeping track of potentially large amounts of data (characterizing records along each path) contribute to the significant execution time of this step. Subsection 4.2.1 evaluates the execution time for calculating the durations in comparison to all other pipeline steps. As an optimization, the implemented algorithm does not perform a single merge step at the very end, i.e. after collecting all paths, but it performs it at every branch point. Duplicate segments can therefore be pruned as early as possible reducing the algorithm's memory footprint.

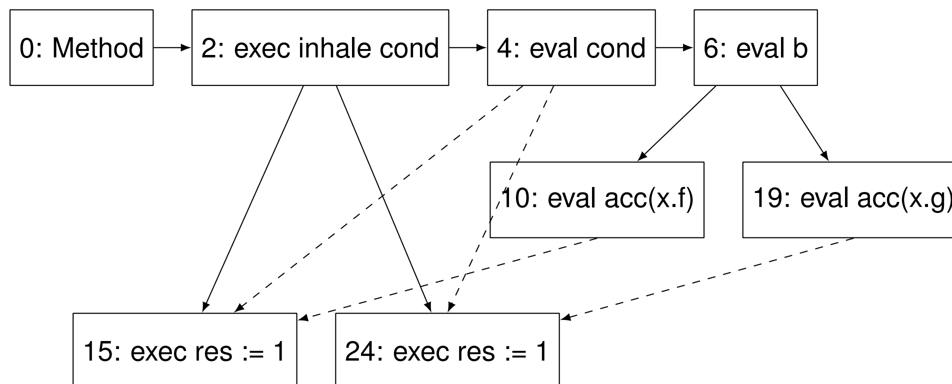


Figure 3.4: Scope representation for the symbolic execution log in figure 2.1. Horizontal arrows denote the sub-scope, vertical arrows the successor, and dashed arrows the inter-level successor relation. ”cond” is used as an abbreviation for `b ? acc(x.f) : acc(x.g)`.

3.4 Scope Creation

The data structure so far consists of data records (now with durations), and open and close scope records. The parent scope relation is only implicitly encoded in the data structure and should be made explicit in this step.

Consider again the example from listing 2.2 and figure 2.1. Figure 3.4 shows the desired scope representation after converting the symbolic execution log. Horizontal arrows denote the sub-scope, vertical arrows the successor, and dashed arrows the inter-level successor relation. A scope level is assigned to each scope and the scope level is increased by one whenever following the sub-scope relation. The root scope, for example scope 0 in figure 2.1, has by definition a scope level of zero. An inter-level successor is a successor that is located on a lower scope level than the current scope. This ensures that every scope either has successors or inter-level successors unless it is the last scope.

In order to achieve the desired output, the scope creation algorithm keeps an array of scopes. The array's index corresponds to the scope level and each level stores the last scope of that level. The algorithm iterates over all records and creates a scope for each data record if it does not exist yet. Listing 3.2 shows in the first function, how the successor and sub-scope relation is established. It first checks whether the newly created scope, `newScope`, should be placed on a level that already has another scope. If there is already a previous scope (line 3), then `newScope` becomes its successor and replaces it in the `scopes` array (on line 6). In the other case, i.e. `newScope` starts a new scope level, there is a case distinction: In case the new scope should be inserted at the root level (line 8) then there are no relations that should be established and the scope is simply inserted into the array at index 0. Otherwise (lines 10–15), the parent scope is

Listing 3.2: Parts of the scope creation algorithm.

```
func addSuccessorLinks(scopes, newScope, scopeLevel) do
  if  scopeLevel < scopes.length then
    let predecessor := scopes[scopeLevel];
    if !predecessor.successors.contains(newScope) then
      predecessor.successors.add(newScope);
    scopes[scopeLevel] := newScope;
  else if scopeLevel == 0 then
    scopes[0] := newScope;
  else
    let parentScope := scopes[scopeLevel - 1];
    if parentScope.hasSubScope then
      insertWithDummyScope();
    else
      parentScope.subScope := newScope;
      scopes.add(newScope);

func addInterLevelSuccLinks(scopes, newScope, scopeLevel) do
  if scopeLevel + 1 < scope.length then
    scopes.slice(scopeLevel + 1, scopes.length)
      .forEach(scope => {
        if !scope.interLevelSucc.contains(newScope) then
          scope.interLevelSucc.add(newScope);
      });
    scopes := scopes.slice(0, scopeLevel + 1);
```

retrieved from the array and its sub-scope is set to `newScope`. However in case the parent scope already has a sub-scope, a dummy scope has to be created[1]. The dummy scope will have the previous sub-scope as well as `newScope` as successors and becomes the sub-scope of the `parentScope`. This indirection is necessary, because only a single scope can be set as sub-scope.

The second function in listing 3.2 shows how the inter-level successor relation is made. It has to be done only when the scope level decreases compared to an earlier iteration. Applied to the example in figure 2.1, this means `scopes` has length four when the scope for `exec res := 1` is created and inserted at level 1. Therefore, the last scopes at levels 2 and 3 should get an inter-level successor. Line 19 of the algorithm hence selects these scope levels and line 22 adds `newScope` as inter-level successor if it is not set yet. Afterwards, line 24 only retains the first `scopeLevel` entries of the array, because the other scopes do not matter anymore for creating scopes for subsequent data records.

In case the log contains branching, the algorithm ensures that each branch gets an independent copy of the `scopes` array. Hence, each branch sets the succes-

[1] The details of `insertWithDummyScope` are omitted in the code listing.

Listing 3.3: Interface of the `mapAfter` iterator function.

```
function mapAfter<T>(
  callbackfn: (
    scope: Scope,
    successorsResult: T[],
    interLevelSuccessorsResult: T[],
    subScopeResult: T) => T): T;
```

sors and inter-level successors on the correct scopes. In terms of the example in figure 2.1, after creating scope 6, the array is duplicated and handed to each branch. As a consequence, the creation of scopes 10 and 19 only modifies the individual branch's copy of the array and when creating the scopes 15 and 24, the array clearly indicates that they are inter-level successors of scope 10 respectively 19.

3.5 Scopes

Scopes, as created in section 3.4, represent an intermediate data structure. It is a DAG-like data structure, where each scope can potentially have another DAG as its sub-scope. The implementation provides several iterator functions to simplify the traversal of the data structure. One such iterator function is `mapAfter`, as defined in listing 3.3. Its use-case is to transform the entire data structure to a result of the generic type `T`. It can be called on any scope, but it is typically invoked on the root scope. `mapAfter` traverses the data structure in a depth-first manner and invokes `callbackfn` for each scope. In addition, `callbackfn` provides the results for the scope's sub-scope, successors and inter-level successors. `mapAfter` is used at several places throughout the Visualizer's pipeline. In particular, it is used for removing certain scopes from the data structure, which is discussed in more details in the next subsection.

3.5.1 Scope Filtering

This subsection discusses one particular scope iterator function: scope filtering based on a predicate. In the current pipeline, this iterator function is used to remove scopes that are marked as being syntactic. All dummy records created by the scope creation algorithm (see section 3.4) are marked syntactic. Furthermore, a symbolic execution framework is free to declare log records syntactic. Scope filtering acts as a case study to demonstrate the involved difficulties of a, at first sight easy, problem.

Recall the visualization of scopes in figure 3.5, which will be used throughout this discussion: Scope 1 has scope 0 as parent scope and scope 4 as sub scope. Furthermore, scope 1 has two successors, namely scope 2 and 3. Scopes 2

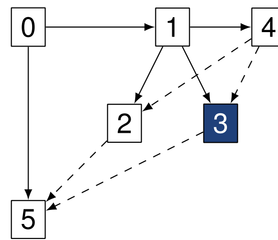


Figure 3.5: Sample of a scope data structure. Scope 1 is sub-scope of 0 and has two successors, namely scopes 2 and 3. Scope 2 has 5 as inter-level successor. Scope 3 is marked for removal.

and 3 each have scope 5 as inter-level successor. Additionally, the notion of scopes filled with blue color is introduced to mark scopes that should be removed.

The problem of scope filtering is as follows: Each scope for which the predicate returns **`false`** should be removed from the data structure, if structurally possible.

The algorithm requires multiple iterations and uses `mapAfter` (see section 3.5) in each iteration to modify the data structure. For each scope, the predicate is invoked to evaluate whether the scope should be kept or filtered out. The predicate's return value for each scope is cached to ensure that the return value does not change from iteration to iteration. Otherwise, the predicate's return value for a particular scope could change in each iteration thus invalidating any guarantees on the number of iterations (as done towards the end of this subsection).

In case a scope should be removed, these four cases[2] can occur:

1. The scope neither has sub-scopes nor successors.
2. The scope has successors but no sub-scopes.
3. The scope has sub-scopes but no successors.
4. The scope has sub-scopes as well as successors.

The following evaluation of these cases only considers changes to the sub-scope, successor, and inter-level successor relations. The implementation changes all inverse relations as well. For brevity, these changes will not be mentioned.

Case 1. Consider the situation as in figure 3.6. The scope with ID 3 should be removed. To do so, scope 1 has to remove scope 3 as a successor and add all inter-level successors of scope 3 (i.e. scope 4) as its own inter-level successors. In addition, scope 2 has to change its inter-level successor from scope 3 to scope 4.

[2]The presence and number of inter-level successors do not matter for the case distinction, because each case can handle any configuration of inter-level successors.

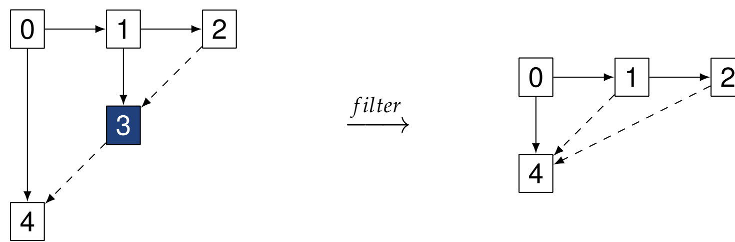


Figure 3.6: Removal of scope 3 corresponding to case 1.

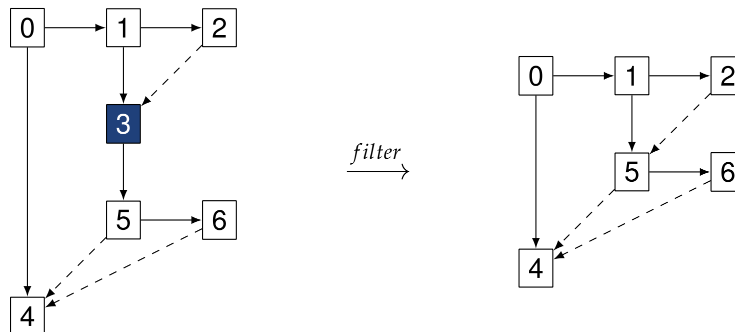


Figure 3.7: Removal of scope 3 corresponding to case 2.

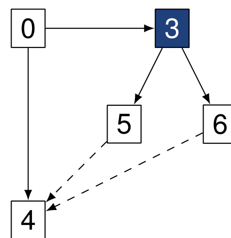


Figure 3.8: Non-permitted removal of scope 3 in a special situation of case 2.

Case 2. Figure 3.7 shows scope 3 which again should be removed but has scope 5 as successor. Scope 1 and 2 have to change their successor respectively inter-level successor from scope 3 to scope 5. Scope 6 is left unchanged, because it does not have any inter-level predecessors.

However, there is a special situation with case 2, in which no scope removal is permitted. Consider scopes as shown in figure 3.8. If scope 3 only had a single successor, removing scope 3 would be possible. However, by having multiple successors, scope 0 would have to change its sub-scope from scope 3 to scope 5 and scope 6. This is not permitted by the data structure and is the same reason for creating a dummy scope during scope creation (see section 3.4). Therefore, the removal of scope 3 is not performed. Nevertheless, scope 3 might be removed in the next iteration of the algorithm.

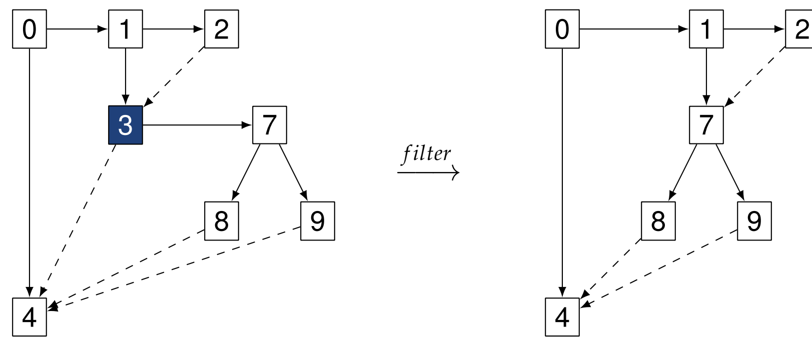


Figure 3.9: Removal of scope 3 corresponding to case 3.

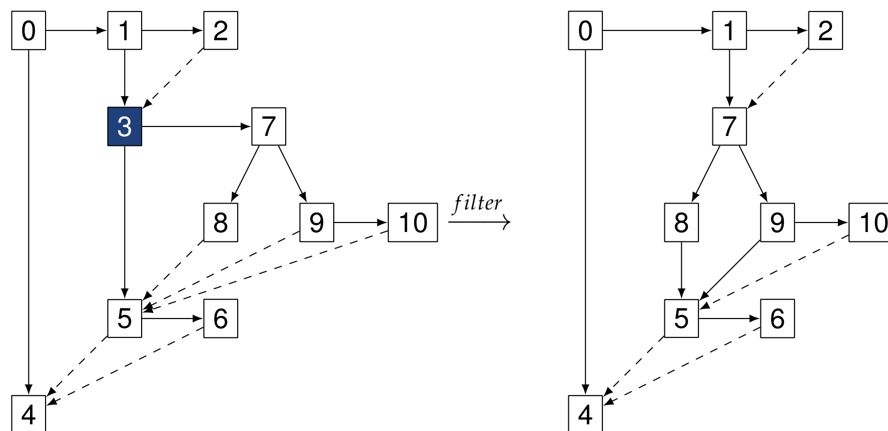


Figure 3.10: Removal of scope 3 corresponding to case 4.

Case 3. In case a scope should be removed that has a sub-scope, then it should be replaced by its sub-scope. This situation is shown in figure 3.9. The scope with ID 3 has a sub-scope consisting of the DAG 7, 8, and 9. To achieve this, the successor of scope 1 has to be changed from scope 3 to scope 7. Furthermore, each inter-level predecessor of scope 3 has to switch to scope 7 as well.

Case 4. Last but not least, case 4 corresponds to removing a node which has successors as well as a sub-scope. This is depicted in figure 3.10. Scope 3 should be replaced by its sub-scope, similarly to case 3. However, the last successors of the sub-scope, in this case scopes 8 and 9, have to be changed as follows: Their inter-level successors now become (direct) successors.

The transformations for removing a scope are mostly limited to the neighboring scopes, i.e. scopes having a reference to the scope that should be removed. The only exception is case 4, where the last successors in the sub-scope of the scope that should be removed are modified as well. However, the presented local transformation rules cannot perform the intended removal if the special situation

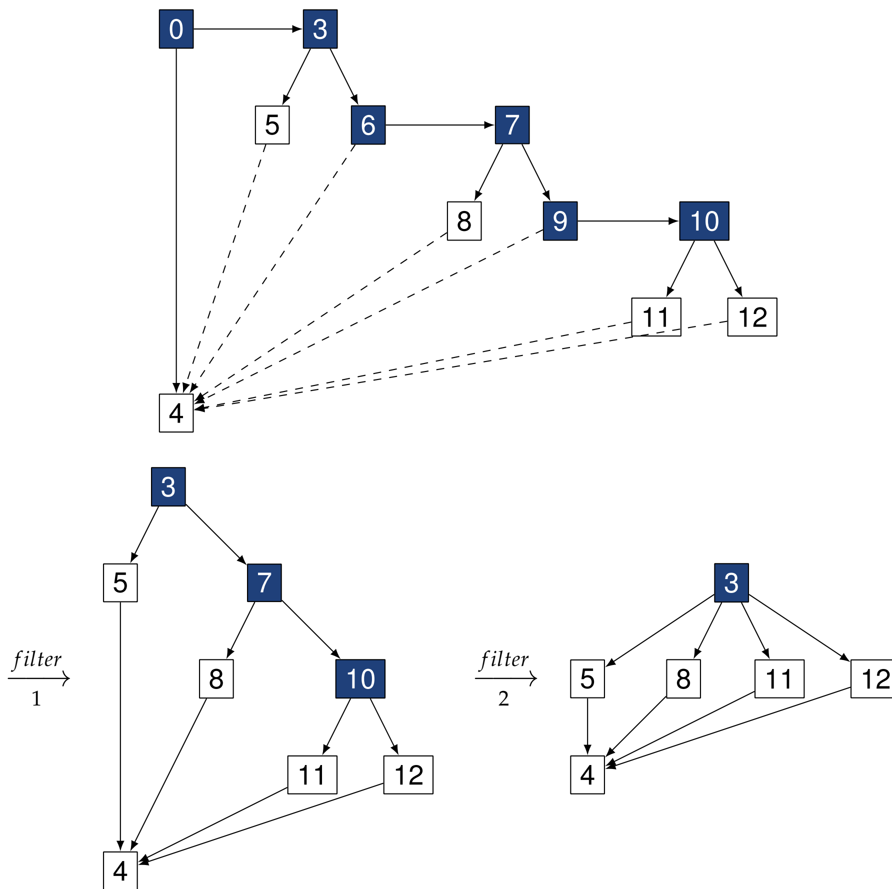


Figure 3.11: Scope data structure needing two filtering passes.

of case 2 occurs. This is the reason why a single iteration of the algorithm is not sufficient in the general case.

Figure 3.11 shows an example that requires more than one iteration over the data structure. After the first iteration, scopes 3, 7, and 10 are still present, because they could not be removed. Nevertheless, the parent scopes of these remaining and still to be removed scopes were removed in the first iteration. Hence in the second iteration, scopes 7 and 10 are successors and no longer roots of their sub-DAGs. Thus, these scopes can be removed in the second iteration.

In fact, it can be shown that only two iterations are necessary in the worst case. A third iteration would not result in any change in the data structure. The interested reader is referred to the detailed analysis of this claim in appendix B.

3.6 Graphical Representation

At the end of the entire pipeline, the resulting scope data structure will be rendered as scalable vector graphics (SVG), allowing it to be displayed in any common web browser. Each scope is mapped to a rectangle and the individual rectangles are linked according to the successor relation defined on scopes. Already short symbolic execution durations lead to a massive amount of data. Thus, several design choices have been made to create comprehensive but at the same time understandable visualizations: The first and most important one, taken from [18], is that the user selects the level of details. This is implemented by initially showing only the top-level rectangle, as an overview, and the user can unfold sub-steps by clicking on the rectangle. Next, a particular level might contain so many rectangles that the user is unable to identify significant ones. Thus, the height of each rectangle is selected based on a user-supplied function, which should increase a rectangle's height based on its significance. Furthermore, various navigation features are supported, for example zooming or isolating a specific rectangle.

Figure 3.12 shows the visualization of the symbolic execution of a Viper program, containing a few branches. The top-level rectangle representing a method was expanded to show its content, which mostly corresponds to revealing the individual statements. The height of each rectangle is selected with respect to its scope's duration, which is the default. This allows users to quickly identify the scope that contributes most to the total execution time. In addition, there is a white hover tip that lists several key metrics for the hovered rectangle, for example the total duration and number of SMT queries. Furthermore, the number of invocations and total duration of some particular algorithms of Silicon are listed in the hover tip. The algorithms for removing permission as well as merging and consolidating symbolic state have been selected, because we have evaluated that they sometimes significantly contribute to the execution time of Viper programs. Thus, having a short summary about their execution in the hover tip allows to quickly detect abnormalities.

Navigation. The Visualizer incorporates several features to facilitate navigation. The first one, clicking to reveal the content of a scope, was already described in this section's introduction. In addition, the visualizer allows dragging and zooming with finger gestures. Zooming is also possible using the plus and minus buttons in the bottom right corner. The initial zoom and pan settings can be restored using the restore button in the same corner. Moreover, shift-clicking a scope will isolate it from all surrounding scopes, allowing a more focused inspection of a particular scope. Lastly, scopes can be alt-clicked for selecting a specific path through the program.

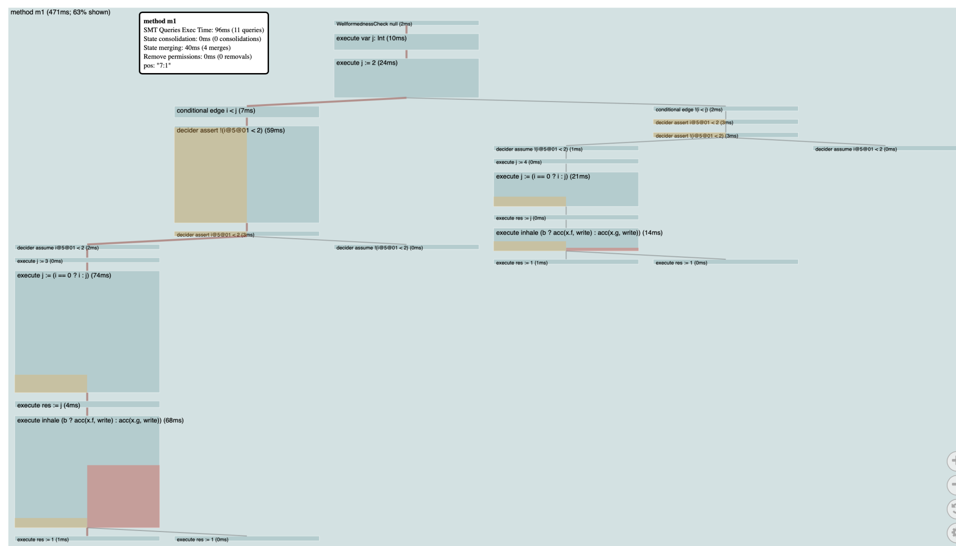


Figure 3.12: Visualization of the symbolic execution of a small Viper program.

Path Selection. Due to the presence of joins, simply selecting a path end does not uniquely identify all scopes on the path. Therefore, the notion of weights is introduced. Weights influence the path selection, which chooses the path with the largest sum of weights. Alt-clicking a rectangle will assign the weight of one to it and allows the user to readjust the selected path by placing additional weights in the graph. An additional advantage of using weights is that the user can select one or multiple path segments of the entire symbolic execution path. A path segment starts at a rectangle with weight one and extends to the next rectangle with weight one, i.e. the selection toggles at every rectangle with weight one.

Figure 3.13 shows as pink filled rectangles the three scopes that were alt-selected, i.e. having weight one. All rectangles that do not have weight one but lie on a selected path segment are rendered with a pink border. As shown, the second path segment extends to the end of the path, because there is no forth rectangle on the path with a weight of one.

The implemented technique of using weights has the benefit that it requires only a few clicks and can iteratively be refined by the user in comparison to defining a path fraction by clicking on all scopes that should be part of it. Furthermore, it is not straight-forward whether sub-scopes are automatically included when the user clicks on their parent scope or not. If not, the user would have to unfold the sub-scopes first and click on all of them.

The user can not only choose between a histogram (as in figure 3.14) or a flame-graph (figure 3.15) to visualize the scopes on the selected path segments but he can also define a function that maps each scope to a value that should be used for the visualization. The histogram includes only scopes that are visible

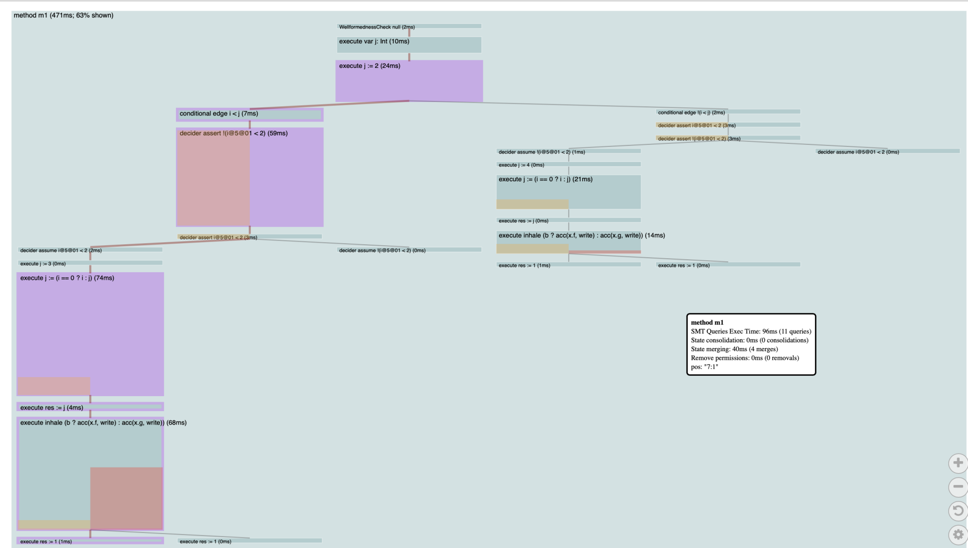


Figure 3.13: Selection of two path segments, drawn with a pink border, in the Visualizer.

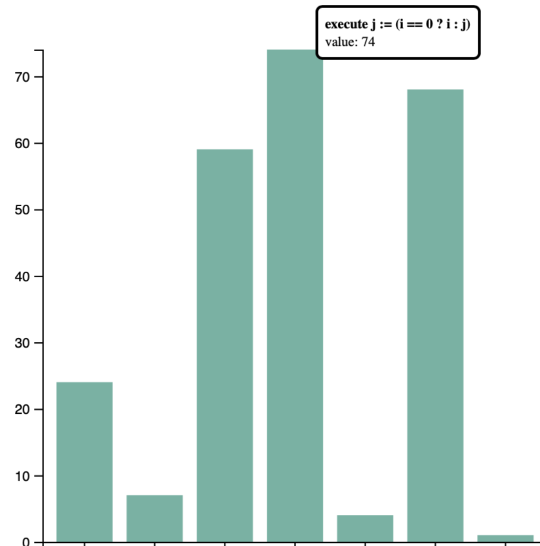


Figure 3.14: Histogram of the execution time for the selected path segments in the Visualizer.

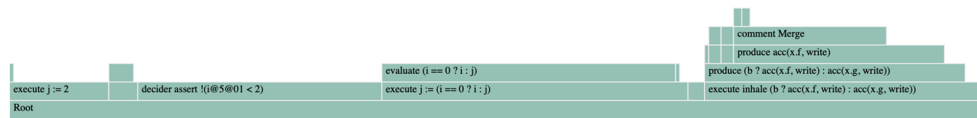


Figure 3.15: Flamegraph of the execution time for the selected path segments in the Visualizer.

and non-expanded. This allows users to steer the scopes that are included in the histogram by expanding or collapsing scopes. The main purpose of the histogram is to give an impression of the evolution of a particular parameter along a path. The flamegraph improves this idea by being able to simultaneously render different scope levels. Therefore, visually collapsed or expanded scopes do not influence the flamegraph, because all scopes along the selected path are rendered. In the histogram, the horizontal axis represents the selected path and the parameter's magnitude is shown vertically. The flamegraph is slightly differently organized, the horizontal axis also shows the ordering of scopes along the path. However, the parameter's magnitude is horizontally rendered as well. Vertically, the flamegraph shows the sub-scope relation.

Although path selection copes with joins by the symbolic execution, the current implementation does not consider scopes on branches that will be joined later on for the histogram as well as the flamegraph. Instead, it will display their parent scopes and omit the sub-scope. The implementation could easily be extended to show scopes in the context of joining as well. However, it is not clear yet whether or how such scopes should be treated. On the one hand, showing the scopes of just one branch might result in wrong impressions, because the work performed on all branches is required at the join point for being able to resume symbolic execution. On the other hand, showing the scopes of all branches that will be joined, in the histogram or flamegraph, would imply some temporal ordering amongst them, because they need to be drawn in some ordering on the x-axis. Although such an ordering actually exists when the symbolic execution engine executes one branch after the other, the execution state will be reset to the state at the branch point for each branch. Hence in terms of state, the individual branches were explored in parallel und should also be drawn as such in the histogram or flamegraph.

Initial experiments with flamegraph visualizations have confirmed that the sequential representation of branches is misleading: Imagine rendering an entire symbolic execution with two branches as a flamegraph. Further assume that the branches were sequentially explored, because otherwise a meaningful visual representation is even harder. The built-in flamegraph engine of Google Chrome, called Trace-Viewer [17], takes a set of start and end timestamps and rearranges them as a flamegraph. Visualizing an entire symbolic execution this way results in

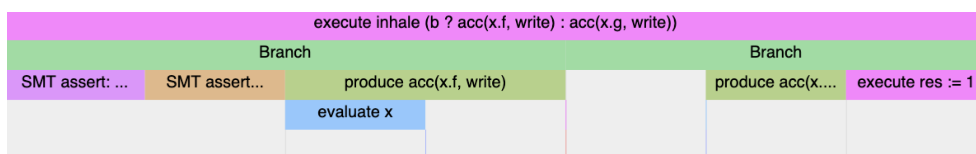


Figure 3.16: Detail of the Trace-Viewer [17] showing branching of a symbolic execution.

a flamegraph that first shows the scope of the first branch with its sub-scopes below it. Figure 3.16 shows a detail of a symbolic execution exploring two branches (shown in green). Following the first branch, the scope of the second branch is rendered, with its sub-scopes below it. Interpreting this flamegraph easily leads to the conclusion, that the second branch follows the first branch. This is correct considering the time axis and the order that the symbolic execution framework has done its work. However, it does not give any insights into the control flow of the execution and might even confuse users, that the state after execution of the first branch was used as input to the second branch. That is why the Visualizer only displays flamegraphs for specific paths and currently omits scopes of branches that get joined.

Configuration. The Visualizer offers in the bottom right corner a settings button that opens a settings popup (see figure 3.17). It currently offers three different settings. First of all, the JavaScript function that is used to define the relative height of rectangles can be changed. By default, the function returns the maximum of three and the scope's duration. Three is an empirically evaluated parameter that ensures a good tradeoff between a truthful visualization of the actual durations as well as readability of scope descriptions for scopes with really short durations.

Secondly, the function that is used to compute the values for the histogram and the flamegraph can be configured. The user can use all fields of a scope, for example a scope's duration (as in figure 3.17), use a statistical parameter of the SMT solver, or access custom data that he added to the log records during the symbolic execution. However, the settings popup does not yet suggest such values. Collecting and displaying a list of these values should be an easy extension of the Visualizer. For now, the user has to know the names of the statistical parameters, for example by looking them up in the log file. The default is using the value `quant-instantiations-delta` from the SMT solver's statistics. In case the function returns **`null`** for a specific scope, i.e. indicating that the scope does not have such a value, an aggregated value will be used instead. The aggregated value is computed by adding up all aggregated values of the sub-scopes. However, there is no difference in computing the aggregated value for scopes that contain joining of symbolic execution branches and scopes that do not.

Thirdly, the user can switch between showing a histogram or a flamegraph for

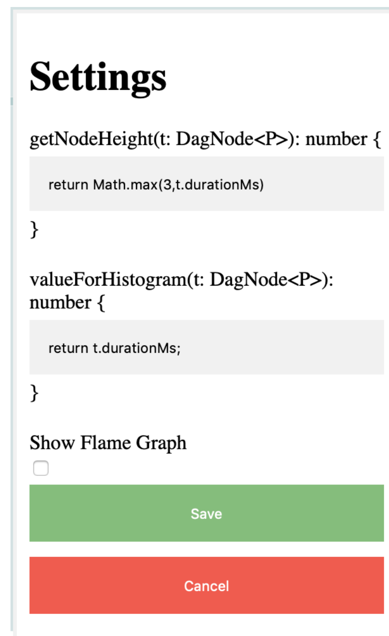


Figure 3.17: Settings popup of the visualizer offering various configuration possibilities.

the selected path. Clicking on the save button will apply the settings and redraw the visualizations, while keeping the same path selection as well as zooming and panning values.

Level of Completeness. Visualizing a symbolic execution log that does not contain the time-wise relevant scopes can lead to wrong impressions of potential bottlenecks, because the actual bottlenecks might be missing. Such incompletenesses in the logged scopes can be detected, because each scope stores its duration. Consider for example the case that a scope takes 100ms but only has a single sub-scope with a duration of 10ms. Calculating the percentage between the sum of sub-scope durations and the scope's duration reveals that only 10% of the scope's duration has been logged. This *level of completeness* is shown by the Visualizer when sub-scopes are revealed. Knowing the *level of completeness* helps in detecting incompletenesses and to indicate that the verification should be rerun, but with additional logging.

3.6.1 JavaScript Console Integration

In addition to the configurations that can be applied using the visual settings popup, the JavaScript console that web browsers typically offer in their developer options can be used as well. There are at least two different use-cases: On the one hand, the default visualization that is created by the pipeline can be adapted.

Listing 3.4: Query in the JavaScript console to count the number of statements in the entire data structure.

```
> root.sum((n) => n.label.startsWith("execute") ? 1 : 0)
< 14
```

Listing 3.5: API documentation of function `sum` displayed in the JavaScript console.

```
> root.sum.help()
< Iterates over the given node and its subDag (i.e.
  successors at the top-most layer are ignored). For each
  considered node, callbackfn will be called to calculate
5 that node's value (not considering any successor /
  subDag).
  The value resulting from a node, is calculated as
  follows: nodeRes = max(callbackfn(node), subDagRes) +
  successorsRes
10 callbackfn: (node: DagNode<P>) => number: function to
  be called for each node
  cacheGet: (node: DagNode<P>) => number: nullable
  function to get cached result
  cacheSet: (node: DagNode<P>, val: number) => void:
15 nullable function to store val in cache
  returns the sum
```

For example, specific scopes can be highlighted or a marker can be inserted on an edge between two scopes if certain properties are fulfilled. On the other hand, queries to calculate a value can be executed on the entire data structure. For instance, the number of branches can be computed, which might give some insights into the overall *shape* of a symbolic execution.

For this reason, the root scope that is currently rendered is attached to the `window` object. The iterator functions discussed in section 3.5 are at disposal and allow the creation of concise queries, as seen in listing 3.4. Furthermore, each iterator function has a help function to print a short documentation (see listing 3.5). Currently, the user has to know what queries are possible and how to create them. Assembling a list of common queries and providing them as a toolbox to choose from would be a great addition to the Visualizer, especially helping new users.

To ease the adaptation of the visualization, scopes provide convenient functions to change a scope attribute, an attribute of a link to another scope, to add a marker on a link with certain attributes, or to add an area on a scope with attributes. Attributes are arbitrary key-value pairs that will directly be mapped to

Listing 3.6: SVG attribute change of a scope rectangle.

```
root.eachBefore(callbackfn);
function callbackfn(scope) {
  let smtExecTime = scope.getTotalSmtQueryExecutionTimeMs();
  if (smtExecTime > 50) {
    scope.setNodeAttribute("opacity", "1");
  }
}
```

SVG attributes. As an example, listing 3.6 sets the opacity of all scopes to 100% if they have an aggregated SMT solver execution time over 50ms.

3.7 Bottlenecks

During the development as well as evaluation of the Visualizer, several bottlenecks have been identified. As already mentioned in section 3.3, calculating each scope's duration based on all program paths from the open scope record to the close scope records contributes most to the execution time of the entire pipeline, despite the employed optimization to prune duplicate path segments as early as possible.

The rendering of scopes has a decent performance, because only a specific scope level is handed to the layout engine to compute the rectangles' positions and sizes. Hence, revealing the sub-scopes by clicking on a scope mostly consists of laying out the sub-scopes and rendering them in their own coordinate system.

Another limitation is stack size of the evaluating JavaScript engine. Although this affects the entire pipeline, if the stack size is exceeded then it typically occurs at the beginning of the pipeline until scopes have been created. After scope creation, the number of elements in the data structure is significantly reduced, because open and close scope as well as data records are combined to a single scope object. Hence, the recursion depth when iterating over the data structure reduces as well. Furthermore, the default stack size heavily depends on the JavaScript engine. Experiments have shown that Google Chrome exposes about the same stack size limit as Node.js uses by default. On the contrary, the Apple Safari web browser allows a stack of more than twice the size of Google Chrome.

The stack size limitation can be relaxed by using Node.js with a manually configured stack size to preprocess the symbolic execution log and create the scope representation out of it. The scopes can then be written to disk and later on imported in the web browser of choice. In addition, this speeds up opening a symbolic execution log, because preprocessing and conversion to scopes has to be done only once.

3.8 Continuous Integration

The continuous integration for the Visualizer consists of a Docker container [22] providing the environment for compilation and testing, which includes Node.js as well as Google Chrome. It can be run on any build server being able pulling and executing images from Docker Hub [21]. The tests feature various unit tests for the different data structure iterators and the scope creation algorithm.

4 Evaluation

First, the new SymbEx Logger is evaluated in terms of performance overhead as well as its extensibility (4.1). It is followed by the Visualizer's evaluation (4.2), which includes a quantitative assessment and several case studies of specific Viper programs.

4.1 SymbEx Logger Evaluation

Although the work of the logger framework during the symbolic execution is kept to a minimum, creating log records, inserting them into the log, and creating a log report at the end of the symbolic execution affects the verification's execution time. Table 4.1 shows the execution times of various Viper programs, which were generated either manually or by different Viper front-ends. These programs were provided by the Programming Methodology group and successfully verify with a timeout of 100s if logging is disabled. Silicon's execution time with and without logging are shown, as well as the file size of the resulting log file. Before verifying these programs, a simple warm up of the Java virtual machine (JVM) was performed by verifying nine rather short Viper programs. Each program, for which the execution time was measured, was verified five times with and without logging. The execution time of the median runs are shown in table 4.1. Six Viper programs resulted in out of memory exceptions when logging was enabled. Thus, the table does not show an execution time and log size for them. This problem has not been addressed in this thesis, but, as mentioned in subsection 2.2.2, it could be mitigated by splitting the log into several parts and reporting them individually.

By default, logging is disabled during a verification. Enabling logging increases the verification duration on average by 61% for the considered Viper programs. However, there are some outliers for which verification becomes faster after en-

Table 4.1: Silicon median execution time without logging, with logging, and resulting log file size for various Viper programs.

Program	median [s]	median with logs [s]	median log size [MB]
composite.vpr	13.88	18.40	4.30
RelAcqDblMsgPassSplit.sil	3.27	2.52	2.50
RelAcqMsgPass.sil	1.19	0.97	1.30
Knuth_shuffle.vpr	4.97	9.12	9.90
ansi-term-external-call.vpr	64.02	154.49	121.40
nested-loops-3.vpr	25.78	33.19	13.60
paths.vpr	16.01	97.38	28.20
ArrayInt_true_13.vpr	30.37	–	–
ArrayInt_true_14.vpr	36.74	–	–
ArrayInt_true_simple_13.vpr	14.53	24.67	64.40
ArrayInt_true_simple_14.vpr	17.44	27.03	76.60
Chromosome_true_13.vpr	19.49	25.93	88.70
Chromosome_true_14.vpr	23.38	34.70	105.00
Chromosome_true_simple_13.vpr	46.76	–	–
Chromosome_true_simple_14.vpr	59.33	–	–
PokerHand_part1_13.vpr	121.23	–	–
PokerHand_part1_14.vpr	109.87	–	–
test-fast.sil	7.70	9.58	1.10
test-slow.sil	14.94	18.22	1.50
0075_AVLTree.nokeys.sil	18.28	46.90	87.90
AVLTree.nokeys.sil	20.71	59.77	100.70
testHistogramFull-old.sil	4.37	5.84	2.90
testTreeWand.sil	17.90	30.21	73.70
first-final.rs.vpr	19.70	21.73	24.60
Knuth_shuffle.rs.vpr	5.51	7.01	8.10
Heapsort_generic.rs.vpr	114.59	116.31	104.30
borrow_first.rs.vpr	2.08	2.76	4.60
Selection_sort_generic.rs.vpr	31.47	39.73	37.70
Binary_search_shared_mono.rs.vpr	21.66	32.62	27.10
Fibonacci_sequence.rs.vpr	7.58	9.42	15.00
Towers_of_Hanoi_spec.rs.vpr	1.67	2.60	3.80
Ackermann_function.rs.vpr	4.77	7.02	10.60
Binary_search_shared.rs.vpr	12.13	16.56	18.40
100_doors_generic.rs.vpr	7.52	11.40	12.60
Selection_sort.rs.vpr	51.81	52.88	46.90
Langtons_ant.rs.vpr	22.11	21.98	31.50

Listing 4.1: Code snippet to add a comment record indicating the beginning of the `removePermissions` algorithm.

```
val rmPermRecord = new CommentRecord(
  "removePermissions", s, v.decider.pcs)
val sepIdentifier = SymbExLogger.currentLog()
  .openScope(rmPermRecord)
```

Listing 4.2: Code snippet to close the scope referenced by `sepIdentifier`.

```
SymbExLogger.currentLog().closeScope(sepIdentifier)
```

abling logging. This partially explains the relative high standard deviation of 96%. In addition, the amount of data that is logged per time unit highly depends on the program. For example, in case the entire verification consists just of a single long SMT query then a single record will be present in the log. Not only will the resulting log file be pretty small but the logging overhead will also be minimal.

Extending SymbEx Logger. One key property of the SymbEx Logger is easy extension to include additional log records. Such an extension might not only be necessary in case Silicon changes but can also be necessary when further insights have to be gathered for a specific input program. Following is an example of adding a record for the `removePermissions` algorithm in Silicon's `QuantifiedChunkSupporter`. It demonstrates that the SymbEx Logger fulfills this requirement. At the beginning of the method `removePermissions`, a new comment record with the comment "removePermissions", the current state, and path conditions is created and inserted into the log (see listing 4.1). This marks the beginning of the `removePermissions`'s scope and hence the scope has to subsequently be marked as closed. Closing the scope is done at the end of the algorithm and is shown in listing 4.2.

Creating, opening, and closing scopes, as demonstrated, is very simple and can be placed anywhere during the symbolic execution. The current member's log can be accessed by calling `SymbExLogger.currentLog()`. However, one has to be careful not to violate the wellformedness property of scopes: Every scope that is opened has to be closed before the parent scope is closed.

In Silicon, this property can easily be maintained thanks to its architecture based on continuation passing. Each symbolic execution operation that opens a scope will close it at the beginning of its continuation. Thus, not only proper scope nesting is achieved but also any state required to close the scope (i.e. `sepIdentifier` in listing 4.2) can be kept locally.

4.2 Visualizer Evaluation

The Visualizer operates on JSON input as specified in section 2.2. Hence, it is agnostic not only to the symbolic execution framework creating the log but also to the source language of the input program to the symbolic execution framework. However, only a logger for Silicon producing the desired log format is currently implemented. The log format is not specially tailored to Silicon and therefore other symbolic execution framework should easily be able to record logs the same way.

4.2.1 Quantitative Evaluation

This subsection evaluates the Visualizer's performance in terms of loading time for various symbolic execution log files that have been collected from the same Viper programs as used in section 4.1. Node.js with a manually configured stack size of its JavaScript engine was used to execute all pipeline steps except the last one (i.e. scope rendering). 8185kB was experimentally evaluated to be the largest stack size for which a *call stack size exceeded* JavaScript exception is thrown when exceeding the stack size. Since the command line option `--stack_size` only configures the size of the stack region that the Chrome V8 JavaScript engine can use without actually increasing it, choosing a larger stack size than 8185kB leads to segmentation faults of Node when the entire stack is filled up. Increasing the actual stack region provided by the operating system (OS) to Node likely requires recompiling Node with a larger stack size configuration. However, this path was not further explored and 8185kB was used as stack size instead. From the 30 Viper programs in table 4.1 that successfully produced a symbolic execution log file, only 16 can be processed by the pipeline without exceeding a timeout of 1h or the stack size. The recursion depth as well as the algorithm to calculate the scope durations could be improved in future work to increase the number of symbolic execution logs that can be processed by the Visualizer.

The last pipeline step, displaying the scopes as SVG, was performed and measured in the Apple Safari web browser. The JavaScript console window was closed while executing this pipeline step, because rendering the SVG can take over four times longer when the console window is visible. The pipeline was run five times for each program and the median execution time for each step was used to compute the average over all programs. Figure 4.1 shows the average execution time for each pipeline step. As already mentioned in section 3.3, calculating the duration for each scope is by far the slowest part of the pipeline and takes about 90% of the total execution time. A more efficient algorithm replacing the current duration calculation would therefore potentially result in a large speed-up.

The other limitation of the Visualizer is stack space. Rewriting the current iterator functions to use a more iterative instead of recursive approach would allow the processing of log files that are currently too large. Some effort has already been

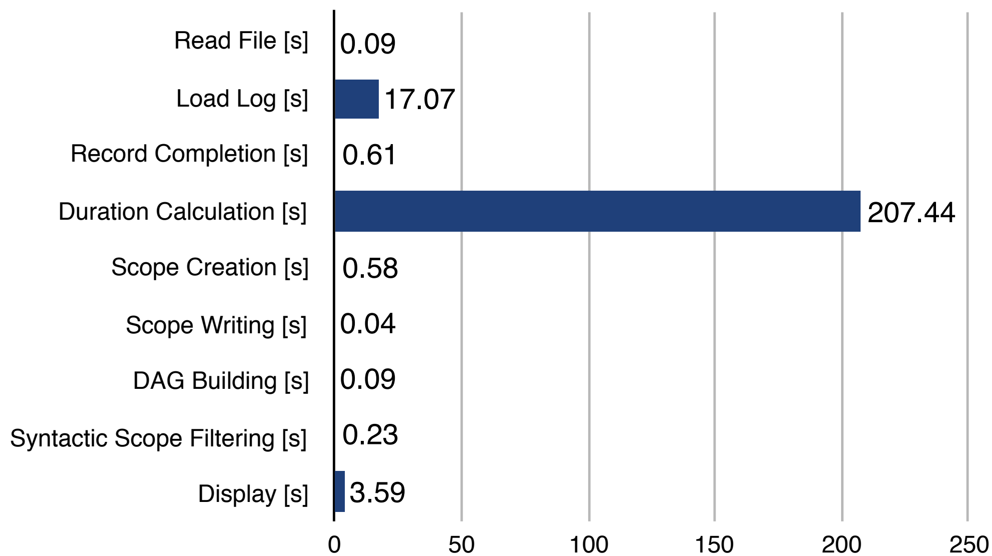


Figure 4.1: Average execution time of each pipeline step based on symbolic execution logs of 16 Viper programs.

made by using more loops and less recursion as well as to cache intermediate values, but there is more work to be done.

In conclusions, the Visualizer provides a decent loading and processing time for a symbolic execution log. Furthermore, the intermediate data representation based on scopes can be exported and later imported. Thus, the most time consuming step of calculating durations can be skipped. The Visualizer is currently limited by the time to calculate durations and the recursion depth of the algorithms needed to traverse the log.

4.2.2 Qualitative Evaluation

The Visualizer's goal is to assist in finding performance problems in the programs that are verified as well as in the symbolic execution framework itself. Hence, this subsection evaluates the Visualizer's usefulness by listing the specific steps that were performed in the Visualizer and the insights that one has gained by doing so. 11 Viper programs have been profiled and corresponding profiling minutes have been created. This subsection discusses four of them that turned out to be interesting. The profiling minutes of the other Viper programs can be found in appendix C. Furthermore, two examples have been in parallel and manually analyzed by Moritz Knüsel [29]. For these examples, the results provided by manual inspection and inspection with the Visualizer will be compared.

Composite.vpr The first considered Viper program is composite.vpr. It was selected, because Knüsel was in parallel analyzing it. Hence, tool support provided by the new SymbEx Logger and the Visualizer can be compared to manual findings. Figure 4.2 shows the profiling minutes for analyzing the symbolic execution of this program. It lists the steps performed in the Visualizer to answer a specific question. In conclusion, symbolically evaluating one of the three postconditions of the method call to `addToTotal` takes up more than 50% of the entire method's execution time. For producing this specific postcondition, 96% of the time is spent in the SMT solver for various SMT queries. Diving deeper into the analysis of the subexpressions shows that the execution time of the postcondition's production is dominated by four SMT solver queries, each taking around 500ms.

Knüsel came, by manual inspection, to the conclusion that Silicon spends most of the verification duration in a loop in the method `removePermissions` in the `QuantifiedChunkSupporter`. Although the SMT solver queries took never the full 500ms in his case, the dominant queries do come from the same algorithm in Silicon. Therefore, the Visualizer points in the same direction as manual inspection to conduct further analyses. The manual inspection took him a couple of days to identify the `removePermissions` algorithm as a potential performance problem. In comparison, analyzing the Viper program with the Visualizer was possible in about two hours.

During the analysis of this Viper program, the logger was not extended yet to record `removePermissions` invocations, as described in section 4.1. Furthermore, the SMT solver's statistical information was not included in the log yet and the Visualizer was not able to display a histogram or flamegraph yet. Hence, steps using these features are not present in the profiling minutes.

Rerunning the verification and analysis with the most recent logger and Visualizer features shows that `removePermissions` is invoked 40 times, making up for 71% of the entire method's execution time. In terms of quantifier instantiations, the method call to `addToTotal` stands out by having a value of over 155'000, which is almost exclusively split among evaluating preconditions (74'000) and postconditions (80'000). This shows that the logger can easily be extended, as well as that the Z3 statistics can contain insightful information. However, it is often not a priori obvious which Z3 metric is most relevant and correlates best with the SMT solver's execution time. Even if a relevant metric was found, its meaning might not directly emerge from its name. Unfortunately, the Z3 metrics are undocumented and it thus remains difficult to interpret them.

In contrast, Knüsel has added log output for quantifier instantiations in Z3 itself and recompiled Z3. This leads to similar results as analyzing the quantifier instantiation metric in the Z3 statistics, because both act as indicators for axiomatisations producing too many instances [19]. To gain further insights into the operations of the SMT solver, the dedicated Axiom Profiler [6] can be used.

Question	Step	Answer
What method has the longest execution time?	Open the Visualizer	Method `addRight` (8241ms)
How much time is spent in SMT solver?	Look at yellow area	78%
How many branches exist?	Count the columns	5
Which stmt has the longest exec time?	Code does not show anything, but switching to relative heights shows it nicely	`addToTotal(this, n.total, X)`
"`addToTotal`" is a stmt, but why does it take so long?	Click on node to reveal content	One node "`method call addToTotal`" is revealed
"`addToTotal`" is a method call, but why does it take so long?	Click on node to reveal content	Several `consume` and one `produce` node significantly contribute to the execution time
What is consumed and produced?	Shift click on method call node to enlarge it	First, arguments are evaluated, then the precondition is consumed (after expanding the definitions) and afterwards the postcondition is produced
What is the largest time contributor?	Look at tallest node	`produce (forall x: Ref :: { (x in X) } (x in X) ==> dom(x.intf) == Set(x) && …`
How much does it contribute?	Manually calculate its fraction to the method call's execution time	52% of method call's execution time
How much does it spend in SMT solver?	Manually calculate the fraction of SMT solver time (2122ms) to the node's duration	96% of produce's execution time
Why?	Expand the "`produce`" node with shift click	The evaluation of the expression takes up most of the time
What steps are involved in the evaluation?	Expand the two enclosed "`evaluation`" nodes	There is an implication which involves joining
What happens in the "join context"?	Expand the "`joining null`" node with shift click	There are two branches. If the left hand side is assumed false, then there is almost no work done. The join overhead is negligible too and does not involve the SMT solver. Almost the entire execution time is spent for the evaluation of the right hand side
Why does the implication's right hand side take so long?	Expand right hand side evalution	The right hand side involves a short-circuiting operator (&&), which leads to a second implication. This takes up most of the execution time
What happens during the implication's evaluation?	Expand "`evaluation v@257@01 …`"	The "join context" takes up almost the entire execution time
What happens in the "join context"?	Expand the "`joining null`" node	Again, the right hand side takes up most of the time (obviously)
Why does right hand side take 2171ms?	Expand "`evaluate (forall z$0 …`"	"`evaluate fdEq(select(out(x.intf), z$0), out_of_node(x, z$0))`" is the major time contributor
What happens in the before mentioned evaluation?	Expand "`evaluate fdEq(select…`"	"`evaluate out_of_node(x, z$0)`" takes 2081ms
Why does "`evaluate out_of_node`" take so long?	Shift click on it to expand it	It again contains a "join context"
What happens in the "join context"?	Expand the "`joining null`" node	The execution time is almost evenly split among "`consume acc(x.left)`", "`consume acc(x.right)`", "`consume acc(x.parent)`", and "`consume acc(x.total)`", each having 2 SMT solver queries taking up 97% of the time
Are the 2 SMT solver queries equally long?	Expand the "`consume`" nodes	no, the first one of the form "`decider assert QA r :: (inv@_x_@01(r) in X@5@01 ? W : Z) - pTaken@_y_@01(r) == Z`" (`_x_` ∈ {233, 238, 243, 248} and `_y_` ∈ {259, 261, 263, 265}) takes between 486 and 527ms long, the second one is of very short duration

Figure 4.2: Profiling minutes for analyzing the symbolic execution of the Viper program composite.vpr

borrow_first.rs.vpr The Viper front-end Prusti [3] was used to generate Viper code, having about 480 lines of code (LOC). As opposed to the previous Viper program, invocations of the `removePermissions` algorithm are not expected to appear in the resulting symbolic execution log, because Prusti does not make use of quantified permissions. Quantified permissions are permissions to a (potentially unbounded) number of heap locations [26].

Figure 4.3 and 4.4 show the steps performed with the Visualizer to answer questions and derive further questions. This Viper program is characterized by its large amount of SMT queries. However, the individual queries are of very short duration, taking on average only 1.9ms and at most 7ms. On the longest path in terms of execution duration, Silicon performs two state consolidations. During a state consolidation, Silicon rewrites its state to a semantically equivalent one to overcome incompletenesses. A state consolidation is cubic in the size of the state modulo SMT queries. Considering the flamegraphs for the time spent in the SMT solver and for the SMT statistics `num-allocs-delta` and `propagations-delta`, the scopes corresponding to the two state consolidations appear as substantial as well.

Silicon offers an option `--enableMoreCompleteExhale` that uses more complex algorithms to handle its state. In consequence of this, state consolidations are no longer necessary. After enabling these more complex algorithms, the same Viper program verifies 23% faster and issues 28% less SMT queries. The average duration per SMT query is unchanged and the maximum duration increases to 24ms, which is still a very short duration. Structurally, the symbolic execution explores the same number of paths and encounters the same amount of branch points independent of enabling or disabling more complex exhales.

Knuth_shuffle.rs.vpr Knuth_shuffle is an implementation of an algorithm to randomly shuffle the elements of an array. Prusti was again used as front-end to compile the 74 LOC to 963 LOC of Viper code. Profiling the corresponding symbolic execution of Silicon is shown in figure 4.5 and 4.6. In general, this example looks similar to the previous Rust example. Both spend over 65% of the total execution time in the SMT solver and issue many small queries. However, Knuth_shuffle contains one relatively big query of 125ms and all other queries take less than 25ms. Nevertheless, 125ms is still a very short duration for an SMT query in comparison to other examples. The flamegraph attracted attention to the evaluation of `0 <= self.val_int`, taking 201ms, for which only really short subevaluations are present in the flamegraph. The same observation can be confirmed by using the Visualizer's main view: The evaluation of `0 <= self.val_int` has two sub-scopes, the first one for evaluating `0` and the second one for evaluating `self.val_int`, each taking zero respectively 1ms. Hence, the created symbolic execution log is incomplete in the sense that not all subevaluations were logged. The Visualizer indicates this by showing a *level of completeness* of zero.

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	`method m_borrow_first$ $foo$opensqu$0$closesqu$` (1911ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	1427ms (75%)
How many SMT queries are there overall?	Look at the hover info	749
How much time is spent for merging state?	Look at the red area and the hover info	326ms (108 merges)
How much time is spent for state consolidation?	Look at the hover info	894ms (3 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
It looks like the execution contains many branches; how many branch points exist?	Enter `` `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` `` in JS CLI	82
How many paths are there?	Enter `` `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` `` in JS CLI	606135
How many statements are there?	Enter `` `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` `` in JS CLI	238
How many SMT queries are on the longest path?	Enter `` `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` `` in JS console	566
How much time is spent in SMT solver on the longest path?	Enter `` `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` `` in JS conosle	1066ms
What is the distribution of the durations on the longest path?	Enter `` `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` `` to select longest path, enter `root.redraw()`, and enter `` `return d.durationMs;` `` as histogram function in the settings popup	2 state consolidations stand out (having an executino time of 232 resp. 376ms)
What scope has the longest duration on the longest path (beside the method itself)?	Enter `` `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method m_borrow”) && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` `` in JS console	496ms: `execute package acc(DeadBorrowToek$(-1), write) && …`
What is the longest duration of an SMT query on the longest path?	Enter `` `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` `` in JS console	7ms
What is the longest duration of an SMT query in the entire method?	Enter `` `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` `` in JS console	7ms

Figure 4.3: Page 1 of the profiling minutes for the symbolic execution of the Viper program borrow_first.rs.vpr

Question	Step	Answer
Does the flamegraph reveal some interesting insights for the longest path?	Compare the flamegraphs obtained via `return t.isSmtQuery ? t.durationMs : null;` and `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` entered as the histogram function in the settings popup	Using only durations of SMT queries changes almost nothing compared to `return t.durationMs;`; same with using the SMT statistical parameter "num-allocs-delta"
How does the "decisions" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.decisions");`	The second state consolidation dominates even more, but this is most likely due to the increasing decisions value
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.propagations-delta");`	The same two state consolidations stand out (144 resp. 655)
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");`	There are around 6 nodes with values between 19 and 25
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");`	There are around 7 significant statements, however only the second state consolidation remains significant. The values for these significant stmts are between 5 and 20 (the entire method has 128)
What is the execution time of the longest path?	Enter `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` in JS console	1436ms

Figure 4.4: Page 2 of the profiling minutes for the symbolic execution of the Viper program borrow_first.rs.vpr

Similarly to the previous Viper program, the effects of enabling more complex exhales on the symbolic execution have been evaluated as well: The symbolic execution time of the longest method more than doubles to 9s. In addition, the time spent in the SMT solver significantly increases to 84% of the method's total execution time. The average duration of an SMT query is 4.4ms and the maximal duration decreases to 27ms. However, the number of SMT queries increases by 44%. Considering the `conflicts-delta` parameter of the SMT statistics, the aggregated value for the entire method increases from 373 to over 9400. This indicates that the SMT solver's search space considerably increases [19].

RSLSpinlock.sil Taken from the Relaxed Separation Logic paper [39], RSLSpinlock is a hand-encoded Viper program from the paper's figure 7. Knüsel has found a matching loop by manual inspection of Z3's log. Furthermore, he was able to support his claim with the Axiom Profiler [6]. The problem of matching loops is that the SMT solver does not terminate: It is able to instantiate axiom after axiom, therefore it looks like the SMT solver progresses. However, the SMT solver recursively instantiates the same axioms. Detecting such instances of recursive behavior is non-trivial and even the Axiom Profiler can only hint at matching loops without absolute certainty.

Knowing that the verification can get stuck in the SMT solver, a timeout of 1000s was used for running Silicon. This timeout results in an exception that will be thrown by Silicon after 1000s. Although the resulting symbolic execution log will be incomplete, the Visualizer is able to process it (as seen in section 3.2).

Profiling the resulting symbolic execution log (figure 4.7) shows that only 2.2% of the entire verification's duration is spent in the SMT solver, somehow contradicting the claim that the SMT solver gets stuck in a recursive loop. Only checks to find

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	method m_Knuth_shuffle$ $knuth_shuffle$opensqu0closesqu$ (4432ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	2876ms (65%)
How many SMT queries are there overall?	Look at the hover info	1191
How much time is spent for merging state?	Look at the red area and the hover info	2802ms (63%) (210 merges)
How much time is spent for state consolidation?	Look at the hover info	0ms (0 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
It looks like the execution contains many branches; How many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	29
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	1799
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	328
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	977
How much time is spent in SMT solver on longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	2426ms (84% of the total SMT solver's execution time)
What is the distribution of the durations on longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select longest path, enter `root.redraw()`, and choose `return d.durationMs;` for histogram	There are 3 statements standing out: "execute inhale acc(usize(_8), write) && … "(244ms) "execute unfold acc(usize(_36.tuple_0))" (375ms) "execute unfold acc(usize(_36.tuple_1))" (635ms) Some strange observations: `evaluate 0 <= self.val_int` takes long (201ms) but has only really short subscopes `execute unfold acc(usize(_36.tuple_1))` is even more extreme, only 4% are shown -> very good visible in flamegraph but not so match in path view
What scope has the longest duration on the longest path (beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method m_Knuth_") && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	635ms
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	prover assert _20@34@01 == $t@267@01 (125ms)
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	125ms

Figure 4.5: Page 1 of the profiling minutes for the symbolic execution of the Viper program Knuth_shuffle.rs.vpr

Question	Step	Answer
What is the longest duration of an SMT query in the entire method except "prover assert _20@34@01 == $t@267@01"?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.label !== "prover assert _20@34@01 == $t@267@01" && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	25ms
Do the SMT statistics show some interesting insights for the longest path?	Compare the histograms obtained via `return t.isSmtQuery ? t.durationMs : null;` and `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` entered in the settings popup	Only the first two above mentioned statements stand out, because due to the strange observation in the third significant statement. After switching to the "num-allocs-delta" SMT statistics, only the first significant statement remains
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.propagations-delta");`	Only first significant statement from above remains
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");`	2 statements with values above 20 dominate: "execute assert ref$m_Knuth_shuffle$$VecWrapper$opensqu$0$closesqu$$_beg_$__TYPARAM__$T$__$_end_$inv__$TY$__ref$m_Knuth_shuffle$$VecWrapper$opensqu$0$closesqu$$_beg_$__TYPARAM__$T$__$_end_$$bool$(_28) && (usize$inv__TY__usize$$bool$(_29) && usize$inv__TY__usize$$bool$(_30))" (value: 50) "execute inhale m_Knuth_shuffle$$$opencur$$opencur$impl$closecur$$closecur$$opensqu1closesqu$$$len$opensqu$0$closesqu$__TY__m_Knuth_shuffle$$VecWrapper$opensqu$0$closesqu$$_beg_$__TYPARAM__$T$__$_end_$$int$(old[l24](_28.val_ref)) == old[l24](m_Knuth_shuffle$$$opencur$$opencur$impl$closecur$$closecur$$opensqu1closesqu$$$len$opensqu$0$closesqu$__TY__m_Knuth_shuffle$$VecWrapper$opensqu$0$closesqu$$_beg_$__TYPARAM__$T$__$_end_$$int$(_28.val_ref))" (value 21)
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");`	3 statements stand out with values above 30: "execute unfold acc(usize(_30))" (value 45) "execute fold acc(usize(_31))" (value 33) "execute fold acc(usize(_6), read$())" (value 36) The entire method has a value of 373
What is the execution time of the longest path?	Enter `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` in JS console	3852ms

Figure 4.6: Page 2 of the profiling minutes for the symbolic execution of the Viper program Knuth_shuffle.rs.vpr

assertion violations have so far been logged as interactions with the SMT solver. The SMT solver internally keeps a stack of assumptions that is used to perform these checks. Therefore, there are further interactions with the SMT solver to push or pop assumptions. These interactions have not been logged so far, because it was assumed that all heavy work is only performed when checking for satisfiability. However, the SMT solver can in principle already perform work when additional assumptions are pushed onto its stack. Knüsel has noticed that matching loops occur while pushing assumptions to the SMT solver, which is unexpected.

To make the logger collect durations for these interactions with the SMT solver as well, a comment record for each push and pop operation has been added for this Viper program. Rerunning verification as well as profiling shows that the last logged operation is a push, taking up 70% of the method's execution time. Considering the histogram for the duration-wise longest path shows that the last push is by far lasting the longest, accounting for over 99.9% of the duration of all

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	“`method lock`” (999026ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	22521ms (2.2%)
How many SMT queries are there overall?	Look at the hover info	196
How much time is spent for merging state?	Look at the red area and the hover info	0ms (1 merge)
How much time is spent for state consolidation?	Look at the hover info	0ms (1 consolidation)
How much time is spent for removing permissions?	Look at the hover info	21669ms (39 removals)
On the level of statements, one large scope stands out, which one is it?	Look at the largest scope	“`execute inhale (forall r$65: Ref :: { (r$65 in tmpRefSet$0) } (r$65 in tmpRefSet$0) ==> acc(AcqConjunct((!true ? down(r$65) : r$65), 1), perm(AcqConjunct(temp(r$65), 1))))`” (971996ms)
Why does it take 97% of the method’s execution time?	Shift click on this scope	The execute scope will appear as the only scope
What happens while symbolically executing this statement?	Click on the “`execute`” scope	One sub-scope “produce” with the same expression as in “`execute inhale`” appears, taking the same amount of time
What happens while symbolically executing the production of the mentioned expression?	Click on the “`produce`” scope	Nothing happens because the log does not contain any subscopes
Does the Z3 statistic about quantified instantiations provide any indication?	Enter `` `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` `` to select longest path, then set histogram function to `` `return t.getValue(“data.additionalData.smtStatistics.quant-instantiations-delta");` `` in the settings popup	No trend becomes visible, e.g. that this property increases along the selected path
What is the execution time of the longest path?	Enter `` `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` `` in JS console	998719ms
How many branch points exist?	Enter `` `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` `` in JS CLI	172
How many statements are there?	Enter `` `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` `` in JS CLI	101
How many SMT queries are on the longest path?	Enter `` `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` `` in JS console	185
How much time is spent in SMT solver on the longest path?	Enter `` `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` `` in JS conosle	22456ms
What scope has the longest duration on the longest path (beside the method itself)?	Enter `` `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method lock") && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` `` in JS console	971996ms
What is the longest duration of an SMT query on the longest path?	Enter `` `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` `` in JS console	1523ms
What is the longest duration of an SMT query in the entire method?	Enter `` `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` `` in JS console	1523ms

Figure 4.7: Profiling minutes for the symbolic execution of the Viper program RSLSpinlock.sil

push operations together. Interestingly, reducing the timeout to 100s leads to a symbolic execution, in which the long running push operation occurs two statements earlier. This is not directly resulting from the lower timeout, because the statement takes over 69s with a timeout of 100s (and will then be killed) and successfully completes after only 78ms when the timeout is set to 1000s. Although this behavior looks non-deterministic, repeated verification leads to similar results for a given timeout configuration.

This example nicely shows that the logger can easily be extended, either to adapt it to non-terminating verifications or to increase the amount of details in the log. However, selecting additional log records is very simple when knowing that pushing assumptions onto the stack can take long. This information is normally not available. Nevertheless, missing long-running steps do not remain unnoticed thanks to the *level of completeness* metric provided by the Visualizer. Adding more records, rerunning the symbolic execution, and using the Visualizer to further narrowing down the missing step can quickly be done. Thus, the Visualizer speeds up the process of finding suspicious operations (in terms of their execution duration), even if different input programs require different techniques or logger additions.

5 Conclusions

The new logger implementation for Silicon heavily simplifies recording a symbolic execution. It does not only produce a log format that is suitable for later processing but also better preserves the relationships of individual log records, in comparison to previous work. Furthermore, the log format is generic, meaning that it is not specifically targeted at Silicon. Other symbolic execution frameworks should also be able to produce logs in the same format and hence could reuse the post-processing pipeline. The simplified logger implementation, together with the unit test infrastructure, eases maintenance and should ensure that the logger can easily be adapted to upcoming changes of Silicon. Lastly, the logger's integration into ViperServer allows various client applications to access the produced log. In comparison to just letting Silicon write a log file to disk, this increases the flexibility of the overall Viper architecture. In particular, it allows plugins for Visual Studio Code [31] to verify programs using Silicon and access the resulting symbolic execution logs. This enables further projects to build upon the symbolic execution log or to integrate the Visualizer as a plugin into Visual Studio Code and ship it together with the Viper IDE [25].

The Visualizer performs offline processing of a symbolic execution log. Part of the processing pipeline is an intermediate data representation called Scopes. Scopes can be exported so that parts of the processing can be skipped when visualizing the same log multiple times. The last pipeline's step creates an SVG from the scopes. It uses the JavaScript library D3.js [9] as well as D3-DAG [11], which adds support for DAGs to D3.js. However, D3-DAG initially only supported drawing circles for each node in a data structure. Therefore, this thesis contributed several new algorithms to D3-DAG, that add support for rectangles as well as for drawing them with different relative heights. These contributions have been part of D3-DAG since version 0.3.0 [2] and an interactive example [1] allows

to experiment with the various configuration options.

The Visualizer was evaluated on several Viper programs. The gained insights have steered the user in the same direction as a manual inspection concluded. Hence, it speeds up the process of finding performance culprits. In fact, the evaluation has shown that using the new SymbEx Logger together with the Visualizer for profiling the symbolic execution of a Viper program reduces the time to find performance culprits from a couple of days to just a few hours compared to a manual inspection. However, as the evaluation has also shown, the initial log output is often not sufficient and additional log records need to be added in order to find the responsible operation in the symbolic execution. Fortunately, these additions correspond to simple and short code segments that have to be inserted into Silicon. After a potential culprit was identified, further analyses are necessary to find and fix problems. These steps are typically very problem-specific. For example, multiple symbolic executions are performed each using a slightly modified version of the input program. Thus, the effects of the modifications on the symbolic execution, including the SMT solver, can be evaluated. Although the Visualizer does not assist in creating these modifications, it speeds up the process of evaluating the resulting symbolic execution logs.

6 Future Work

In this chapter, we briefly discuss some topics that could be explored in future work. Firstly, the performance limitations of the new SymbEx Logger as well as the Visualizer could be addressed. This mainly increases the maximal symbolic execution time of Viper programs that can be profiled.

Secondly, additional Viper programs could be evaluated. This might reveal not only similarities according to which these programs could be categorized but also visualization features that are currently missing in the Visualizer. Besides adding new features, the design of the existing visualization features could be improved, especially in terms of readability of labels in presence of many rectangles. Improving the design might include switching to a different layout framework, because the current choice turned out to require a lot of work to achieve a desired placement of rectangles.

Next, the current Visualizer analyzes SMT queries and is capable of showing the distribution along a path based on their duration or a statistical parameter from the SMT solver. However, the similarity of SMT queries is not evaluated at all. We estimate that not only SMT queries on the same path but also on different paths might expose similarities. Identifying almost identical SMT queries might be a good indicator for repeated work by the symbolic execution engine. The identified queries might therefore be a performance culprit. Furthermore, analyzing the differing parts of two SMT queries that are almost identical but have significantly different execution times might provide insights about why one of them takes so much longer.

Lastly, Bornholt and Torlak [8] have explored the approach of calculating a score for each symbolic execution step in their symbolic profiler, SymPro. The score indicates the potential of the step being a performance culprit and is based on five different metrics. Based on the score for each symbolic execution step, SymPro

compiles a list with the highest ranking steps. A similar approach might be a great addition to the Visualizer. The score allows to give weight to different metrics and could provide additional insights. The most difficult part of implementing such a technique in the Visualizer is the definition of the metrics that should be considered. Once defined, the function calculating the score for a given scope can simply be used as the function computing the rectangles' height in the settings popup (see section 3.6). Thus, the rectangles' height will no longer be chosen according to their duration (today's default) but to their score. The highest ranked scope is therefore as visible as the scope with the longest duration today. Besides defining such a function, no changes to the Visualizer are required. In particular, the Visualizer already allows to specify custom functions to calculate the rectangles' height as well as the values that are used in the histogram and flamegraph.

A

Technical Details

A.1 SymbEx Logger Configuration

By default, Silicon does not collect a symbolic execution log, because, as evaluated in section 4.1, enabling the SymbEx Logger increases the verification's execution time on average by 61%. Therefore, two command line arguments have been added to configure the creation of a symbolic execution log:

writeLogFile Specifies that symbolic execution logs should be gathered and will create a report of type *ExecutionTraceReport*, containing the log, when the verification has ended.

logConfig Expects a path to a log configuration file. The configuration file specifies not only which records should be included or ignored in the log but also which additional data, such as heap information or path conditions, should be logged.

A.2 SymbEx Logger Unit Tests

Simply comparing the symbolic execution log of a Viper program to an expected log is not robust at all. The test will already fail when the logger is extended to log a single additional scope, as done for example in section 4.1. Therefore, the comparison is weakened to accept the actual symbolic execution log if it contains at least the same content as the expected log.

Consider the short Viper program in listing A.1. This program is symbolically executed with Silicon as part of the unit test. The resulting symbolic execution log is then converted to a simple text-based representation, as seen in listing A.2. The text-based representation lists data and branching records with a certain

Listing A.1: Short Viper program to test the resulting symbolic execution log.

```
method test(b: Bool, x: Ref)
{
  var a: Int
  inhale (b ? acc(x.f) : acc(x.g))
  a := 1
}
```

Listing A.2: Text-representation of the symbolic execution log of the Viper program in listing A.1. Note that line 6 uses `cond` as an abbreviation for `(b ? acc(x.f, write) : acc(x.g, write))`.

```
method test
  WellformednessCheck null
  execute var a: Int
  execute inhale (b ? acc(x.f, write) : acc(x.g, write))
    produce (b ? acc(x.f, write) : acc(x.g, write))
      conditional expression cond
        evaluate b
        decider assert !(b@82@01)
          prover assert !(b@82@01)
        decider assert b@82@01
          prover assert b@82@01
        Branch b@82@01:
          comment: Reachable
          decider assume b@82@01
          produce acc(x.f, write)
            evaluate x
            evaluate write
            comment Merge
              single merge  <= x@83@01.f -> $t@85@01 # W
              decider assume x@83@01 != Null
          execute a := 1
            evaluate 1
        Branch !(b@82@01):
          comment: Reachable
          decider assume !(b@82@01)
          produce acc(x.g, write)
            evaluate x
            evaluate write
            comment Merge
              single merge  <= x@83@01.g -> $t@86@01 # W
              decider assume x@83@01 != Null
          execute a := 1
            evaluate 1
```

Listing A.3: Text-representation of the expected symbolic execution log of the Viper program in listing A.1.

```
method test
  execute var a: Int
  execute inhale (b ? acc(x.f, write) : acc(x.g, write))
    Branch b@82@01:
      comment: Reachable
      execute a := 1
    Branch !(b@82@01):
      comment: Reachable
      execute a := 1
```

indentation level. Open and close scope records are used to compute the indentation level but otherwise do not appear in the text-based representation. The text-based representation is very similar to how the old SymbEx Logger collected records. In fact, it suffers from the same problem as shown in section 2.1: The executions of the assignment statement `a := 1` appear on the lines 21 and 32 in listing A.2. They are more indented than the execution of the **`inhale`** statement on line 4. Therefore, it looks like the executions of the assignment statement are sub-steps of executing the **`inhale`** statement. As discussed, such a representation of the symbolic execution would not be suitable as input to the Visualizer and motivated the creation of the new SymbEx Logger. Nevertheless, this representation can be used to compare two symbolic execution logs. Although some information is lost during the conversion to the text-based representation, it is a compact and human-readable representation of the symbolic execution log. This is especially handy when a unit test fails and the differences need to be analyzed.

To improve the robustness and be resilient also to the removal of some records, the expected symbolic execution log only contains a subset of the currently logged records. The subset consists of records representing Viper members, i.e. procedures and predicates, executions of statements, branching and joining. Listing A.3 shows the text-representation of the expected symbolic execution log for the same Viper program (listing A.1). For example, evaluations or interactions with the SMT solver, i.e. `prover assert` entries, are not part of the expected symbolic execution log. We estimate that the selected subset is a good tradeoff between robustness and detecting regressions.

The comparison of an actual and expected symbolic execution log is currently done as follows: Each line in the text-representation of the actual log has to be more indented than the corresponding line in the expected output if it exists. This is a simplification of checking that a record was logged in the correct scope. Therefore, future work could improve the check by comparing the stack of open scopes at each line in the text-representation of the actual log against the corresponding stack in the expected symbolic execution log.

B

Scope Filtering Analysis

This appendix gives an intuition to why the scope filtering algorithm only requires two iterations and performing a third iteration would not result in any further change to the data structure.

Assume an arbitrary scope data structure with some scopes being marked for removal. By applying one of the rules presented in subsection 3.5.1, each marked scope can be removed in the first iteration, if structurally possible. Only marked scopes that are the root of a sub-DAG and have at the same time at least two successors cannot be removed in the first iteration. Figure B.1 shows a scope data structure that corresponds to this special situation, because scope 3 is marked, has two successors, and is the root of the sub-DAG formed by scopes 3, 5, and 6.

Due to the fact that `mapAfter` is used in each iteration to traverse the scopes, scopes will be visited in post-order depth-first manner. Hence, when visiting a scope x in the first iteration corresponding to the special situation, its successors have already been visited. Therefore, only its parent scope y or predecessors can change between visiting x during the first iteration and the start of the second iteration.

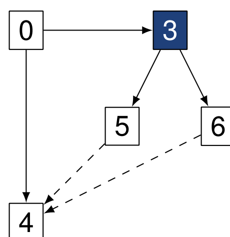


Figure B.1: Scope data structure that does not permit the removal of scope 3.

One can now consider four different situations for the second iteration:

1. x lost its parent scope and got instead predecessors. As a consequence, x is no longer classified as a special situation, because it is no longer root of a sub-scope. x can be removed in the second iteration by applying the rule of case 2 (as defined in subsection 3.5.1).
2. x got a new parent scope z compared to the initial data structure and z is marked. This case can never occur, because it requires that z had a sub-scope in the original data structure containing y and x. Hence, only rules 3 and 4 are applicable to y and z. These two rules do not include any special situation that does prevent the removal of a scope. That is why the existence of z after the first iteration implies that z is not marked, which corresponds to the next situation.
3. x got a new parent scope z compared to the initial data structure and z is not marked. Thus, x still corresponds to the special situation and no change occurs. Any other marked scope has to change in the entire data structure (i.e. correspond to the first situation) in order that a second iteration is even necessary.
4. x still has the same parent scope y. This corresponds to the previous situation and no change occurs.

In conclusion, only the first situation results in a change to the data structure in the second iteration. By application of rule 2, x will be removed in the second iteration. Because x did initially correspond to the special situation (otherwise it would have already been removed in the first iteration), x does only have successors and no sub-scope.

In the third iteration, it is guaranteed that no change to the structure happens. The only candidate scopes that could be removed have to be marked, correspond to the special situation, and have a parent scope that has changed in the second iteration. However, such candidate scopes do not exist, because the second and third condition cannot simultaneously be met: In order for the parent scope to have changed in the second iteration, the old parent scope itself had to correspond to the special situation. By definition of the special situation, such a scope cannot have a sub-scope, hence it cannot have been the parent scope of the candidate scope. Thus, no scope can be removed in the third or any subsequent iteration.

C

Profiling Minutes

In addition to subsection 4.2.2, this appendix provides profiling minutes and conclusions for the symbolic execution of further Viper programs.

Slow.sil Moritz Knüsel has created two versions of the same Viper program that only syntactically differ in a macro. However, the verification's duration is significantly influenced by this macro. Slow.sil takes more than 15s to symbolically execute. In contrast, the faster variant verifies in less than 9s. Figure C.1 shows the profiling minutes for the Viper program slow.sil. The symbolic execution spends 88% of the execution time in the SMT solver. Continuously unfolding the duration-wise longest scope reveals an invocation of the `removePermissions` algorithm taking 1s, which consists of two significant SMT queries of each 500ms. An individual invocation of the algorithm and the SMT queries are rather short. However, the sum of all `removePermissions` invocations and SMT queries significantly influence the overall method's symbolic execution time.

Interestingly, the user's attention is not steered by the Visualizer towards the mentioned macro. Therefore, the logger was extended to include data about the path conditions, state, and heap. After rerunning the symbolic execution of slow.sil and the faster variant of it, a second analysis has been performed. Additional metrics have been evaluated in direct comparison to the faster variant, as seen in figure C.2.

In conclusions, the faster variant has 27% less SMT queries. When considering only the `WellformednessCheck` scope, in which the macro is used, the faster variant also has 28% less entries in the path conditions. The number of store entries is roughly the same for both program variants when considering the entire symbolic execution as well as just the `WellformednessCheck`. However, looking at the maximal sum of ”oldHeap” entries on any path of the symbolic execution,

Question	Step	Answer
What member takes the longest?	Load genericNodes.json	Method “link_DAG__fnext” takes over 15s
Why does the method take so long?	Select the method “link_DAG__fnext”	“WellformednessCheck” uses most time (over 15s)
What part of Wellformedness Check takes so long?	Click on the “WellformednessCheck” node	Last postcondition in “WellformednessCheck” takes up 89% of the time
How much time is spent in SMT solver?	Hover over the last postcondition	14 of 15.7s with 152 queries
Is there some pattern among SMT queries?	Set `SHOW_AGGREGATED_HISTOGRAM` to false	
	Enter `root.transformLongestPath((n) => n.getMaxPathValue(true, (currentNode) => currentNode.durationMs), (n) => {n.weight = 1; return n;})` and `root.redraw()` in JS console	Some queries reach high “num-allocs-delta” values and queries with large values show some increasing trend. The maximum is at 292671
Are there other interesting SMT solver statistical parameters?	Look at JSON data to see all available keys	
	Choose the parameter “max-memory”	An increasing trend is shown
	Choose the parameter “quant-instantiations-delta”	The larger values appear at the same place and with similar shape as when selecting “num-allocs-delta”
	Choose the parameter “mk-bool-var-delta”	The larger values appear at the same place and with similar shape as when selecting “num-allocs-delta”
What SMT query contributes the most in terms of “num-allocs-delta”, “quant-instantiations-delta”, and “mk-bool-var-delta”?	Hover over tallest bar in histogram	“prover assert QA r :: inv@108@01(r) in g@3@01 ==> 1/2 - pTakenn@109@01(r) == Z”
Is there a large node contributing?	Follow the longest path and select the tallest node	Many joins happen, at the end there’s “removePermissions” with 1s having 2 assertions of each 500ms (similar to the above SMT query): “decider assert QA r :: inv@161@01(r) in g@3@01 ==> 1/2 - pTaken@162@01(r) == Z” “decider assert (from@4@01 == from@4@01 ? W : Z) - pTaken@163@01(from@4@01) == Z”

Figure C.1: Profiling minutes for analyzing the symbolic execution of the Viper program Slow.sil

slow.sil has twice as many as the faster variant. This indicates that the syntactical difference of the macro results in a different symbolic heap for the two programs. In addition, the symbolic heap difference results in a different number of SMT queries that are issued by Silicon.

RelAcqDblMsgPassSplit.sil As seen in the profiling minutes (figures C.3 and C.4), a relative low amount of time is spent in the SMT solver in comparison to all other analyzed Viper programs. The verification of the considered method is very short. Nevertheless, the number of branch points as well as resulting paths is very high, especially in the ratio to the number of statements. However, the duration-wise longest path still contributes 60% to the total execution time.

AVLTree.nokeys.sil Figures C.5 and C.6 contain the profiling minutes for the Viper program AVLTree.nokeys.sil. The symbolic execution performs many but small SMT queries, that last on average only 2.7ms. Interestingly, there is a `decider assert` taking over 1.2s. However, during the `decider assert` only 4ms are spent in the SMT solver. Looking at Silicon’s source code reveals that the procedure `deciderAssert` first checks whether a term is known to be true before querying the SMT solver. Hence, iterating over all assumptions in the path conditions takes 1.2s, indicating that the path conditions are very complex. The number of branch points as well as the huge amount of paths point in the same di-

Question	Step	Answer
What's the number of branches in the "link_DAG__fnext" method?	Execute `root.getSubDagWidth()` in the JS console	14
What's the result for the same command in fast.sil?	Execute `root.getSubDagWidth()` in the JS console	14
How many "decider assert" nodes exist in the method?	Execute `root.sum((n) => n.label.startsWith("decider assert")?1:0)` in the JS console	182
What is the result for the same command in fast.sil?	Execute `root.sum((n) => n.label.startsWith("decider assert")?1:0)` in the JS console	132 (27% less than in slow.sil)
Can a similar difference be observed in terms of "removePermission" nodes?	Execute `root.sum((n) => n.label.startsWith("comment removePermissions")?1:0)` in the JS console	25
What is the result for the same command in fast.sil?	Execute `root.sum((n) => n.label.startsWith("comment removePermissions")?1:0)` in the JS console	25
Why does slow.sil have more SMT solver queries while having the same amount of "removePermissions" nodes?	Add "store", "oldHeap", and "pcs" to log and manually save report, because ViperServer cannot handle the report size; Look at the "WellformednessCheck" node	"store" has 3 entries: "g", "from", and "to" "pcs" has 25 entries
	Add "store", "oldHeap", and "pcs" to log and manually save report, because ViperServer cannot handle the report size; Look at the "WellformednessCheck" node	"store" has 3 entries: "g", "from", and "to" "pcs" has 18 entries (28% less than in slow.sil)
Do the path conditions explode?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("pcs") == null) { return 0; } return n.data.additionalData.get("pcs").length ; })`	59539 (60218 when executing the same query on the method's node)
What's the result for the same command in moritz/fast.sil?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("pcs") == null) { return 0; } return n.data.additionalData.get("pcs").length ; })`	48106 (48535 when executing the same query on the method's node) (19% less than in slow.sil)
Does "store" explode?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("store") == null) { return 0; } return n.data.additionalData.get("store").length; })`	1911 (2063 when executing the same query on the method's node)
What's the result for the same command in moritz/fast.sil?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("store") == null) { return 0; } return n.data.additionalData.get("store").length; })`	1902 (2045 when executing the same query on the method's node)
Does "oldHeap" explode?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("oldHeap") == null) { return 0; } return n.data.additionalData.get("oldHeap").length; })`	1014 (1014 when executing the same query on the method's node)
What's the result for the same command in moritz/fast.sil?	Shift click on the "WellformednessCheck" node and execute `root.getMaxPathValue(false, (n) => { if (n.data == null) { return 0; } if (n.data.additionalData == null) { return 0; } if (n.data.additionalData.get("oldHeap") == null) { return 0; } return n.data.additionalData.get("oldHeap").length; })`	504 (504 when executing the same query on the method's node)

Figure C.2: Profiling minutes for a second analysis of Slow.sil including a direct comparison to a very similar but faster variant of the program.

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	Method “thread0” (332ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	146ms (43%)
How many SMT queries are there overall?	Look at the hover info	85
How much time is spent for merging state?	Look at the red area and the hover info	12ms (15 merges)
How much time is spent for state consolidation?	Look at the hover info	0ms (0 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
It looks like the execution contains many branches; how many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	160
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	1129617
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	31
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	42
How much time is spent in SMT solver on longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	68ms
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select the longest path, enter `root.redraw()`, and choose `return d.durationMs;` as the histogram function in the settings popup	One statement stands out: “execute v := a.val” (70ms) It only has “evaluate a.val” as a sub-scope, which also takes 70ms but no further subscopes are visible in the flamegraph
What scope has the longest duration on the longest path (beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method thread0") && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	70ms
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	3ms
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	3ms

Figure C.3: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program RelAcqDblMsgPassSplit.sil

Question	Step	Answer
Does the flamegraph reveal some interesting insights for the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` `` in the settings popup and analyze the flamegraph	2 different statements stand out with values above 1000: "`execute assert acc(l.init, wildcard) && acc(l.acq, wildcard)`" (value 1400) "`execute inhale !is_ghost(a) && heap(a) == 0 && …`" (value 3355)
How does the "decisions-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.decisions-delta");` ``	1 statement is significant with value 33: "`execute inhale !is_ghost(a) && heap(a) == 0 && …`"
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.propagations-delta");` ``	2 statements with values above 10 stand out: "`execute assert acc(l.init, wildcard) && acc(l.acq, wildcard)`" (value 12) "`execute inhale !is_ghost(a) && heap(a) == 0 && …`" (value 26)
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");` ``	There are a couple of statements and decider assertions that stand out, but none of them is above the value 11
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");` ``	There are a couple of statements and decider assertions that stand out, but none of them is above the value 8
What is the execution time of the longest path?	Enter `` `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` `` in JS console	200ms

Figure C.4: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program RelAcqDblMsgPassSplit.sil

rection. Before exploring ideas how to speed up the `deciderAssert` procedure, a second profiling should be performed for analyzing the path conditions.

100_doors_generic.rs.vpr This as well as the next three programs have been compiled from Rust sources using the Viper front-end Prusti [3]. By spending about 78% of the verification's execution time in the SMT solver, this program is dominated by SMT queries. There are many but again small SMT queries having an average of just 2.6ms and 41ms as maximum. Considering the execution time as well as several SMT statistics, two statements stand out. These two statements apply a syntactically long magic wand assertion [35]. Magic wands have not been further considered in this thesis.

Ackermann_function.rs.vpr This Viper program was profiled as seen in figures C.9 and C.10. It is as well dominated by the SMT queries, taking up 78% of the total verification's execution time. All SMT queries are of very short duration, taking on average only 1.8ms. Even the SMT query with the longest duration spends only 21ms in the SMT solver. In terms of duration, the flamegraph does not hint at any significant operation. Considering various SMT statistical parameters, several **`fold`** or **`unfold`** scopes appear as significant. In addition, these scopes contain significant merge operations that combine symbolic state information about equal heap resources. In fact, all merge operations of the method together contribute 78% of the execution time. It is slightly more than what is spent in the SMT solver. This indicates that most SMT queries have been issued by Silicon's state merging algorithm.

Fibonacci_sequence.rs.vpr Figures C.11 and C.12 show the profiling minutes for the Viper program Fibonacci_sequence.rs.vpr. It looks very similar to the previ-

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	method “`remove`” (8166ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	5037ms (62%)
How many SMT queries are there overall?	Look at the hover info	1887
How much time is spent for merging state?	Look at the red area and the hover info	701ms (9%) (256 merges)
How much time is spent for state consolidation?	Look at the hover info	0ms (0 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
It looks like the execution contains many branches; how many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	176
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	40240
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	113
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	220
How much time is spent in SMT solver on the longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	515ms
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select longest path, enter `root.redraw()`, and choose `return d.durationMs;` for histogram	There are four statements with values above 100ms; By far the largest contribution comes from “`execute close(r)`” (1456ms) which includes a decider assertion “`decider assert this@723@01 == m@801@01`” taking 1292ms. The decider assertion only has one small SMT query of 4ms
What scope has the longest duration on the longest path (beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (!n.label.startsWith("method remove”) && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	The above mentioned “`execute close(r)`”
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	6ms
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	57ms

Figure C.5: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program AVLTree.nokeys.sil

Question	Step	Answer
Does the flamegraph reveal some interesting insights for the longest path?	Compare the flamegraphs obtained via `return t.isSmtQuery ? t.durationMs : null;` and `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` entered as histogram function in the settings popup	3 other statements stand out with values above 20000: "execute unfold acc(valid(this))" "execute n1,r := pruneMax(this.left)" "execute unfold acc(valid(r))"
How does the "decisions-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.decisions-delta");`	It roughly has the same shape and same significant statements as considering num-allocs-delta
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.propagations-delta");`	In terms of "propagations-delta", 3 statements are significant: "execute bf := getBalanceFactorI(this, rd)" (value 195) "execute nl, r := pruneMax(this.left)" (value 301) "execute unfold acc(valid(r))" (value 623)
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");`	2 statements mainly stand out: "execute unfold acc(valid(this))" (value 17) "execute unfold acc(valid(r))" (value 17)
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");`	2 statements mainly stand out: "execute unfold acc(valid(this))" (value 59) "execute unfold acc(valid(r))" (value 48)
What is the execution time of the longest path?	Enter `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` in JS console	2210ms

Figure C.6: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program AVLTree.nokeys.sil

ously analyzed program and is as well dominated by SMT queries. Again, it looks as if most SMT queries have been issued during an execution of the state merging algorithm.

Towers_of_Hanoi_spec.rs.vpr The profiling minutes in figures C.13 and C.14 show again that a majority (71%) of the verification's duration is spent in the SMT solver. The flamegraph based on the duration of the individual scopes steers the attention to three statements. However, all of them last less than 100ms and thus do not significantly contribute to the total symbolic execution time. As for the other Viper programs created by the Prusti front-end, the only potential optimization opportunity might be located in the state merging algorithm. These have all been duration-wise relatively short programs though. Therefore, further analyses should be conducted with programs having longer verification durations to confirm that the same observations regarding state merging apply.

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	"m_100_doors_generic$ $doors1$opensqu0closesqu$" takes 7422ms
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	5780ms (ca. 78%)
How many SMT queries are there overall?	Look at the hover info	2200 queries
How much time is spent for merging state?	Look at the red area and the hover info	4834ms (ca. 65%)
How much time is spent for state consolidation?	Look at the hover info	776ms (3 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
It looks like the execution contains many branches; how many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	73
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	10677
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	624
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	980
How much time is spent in SMT solver on the longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	2897
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select longest path, enter `root.redraw()`, and choose `return d.durationMs;` as the histogram function in the settings popup	In the flamegraph, 2 statements are significant (each taking 352 resp. 398ms), both of them are dominated by their state consolidations
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	41
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	41

Figure C.7: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program 100_doors_generic.rs.vpr

Question	Step	Answer
Does the histogram reveal some insights for the longest path?	Compare the flamegraphs obtained via entering `return t.isSmtQuery ? t.durationMs : null;` and `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` as the histogram function in the settings popup	The two significant statements from before appear as significant in both versions and have similar shape
How does the "decisions" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.decisions");`	One statement appears to be significant
How does the "propagations" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.propagations");`	One statement appears to be significant
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.propagations-delta");`	"execute apply acc(DeadBorrowToken${23})..." and "execute apply acc(DeadBorrowToken$(25))..." appear to be significant
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");`	The 2 so far significant statements do also appear but not as the most significant ones. There are about 5 significant statements in total considering this statistical parameter
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");`	The 2 most significant statements are again "execute apply acc(DeadBorrowToken${23})..." and "execute apply acc(DeadBorrowToken$(25))..." The first statement has a much larger value (483) compared to the second significant statement (79)

Figure C.8: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program 100_doors_generic.rs.vpr

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	"`m_Ackermann_function$ $ack2$opensqu$0$closesqu$`" takes 3168ms
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	2474ms (78%)
How many SMT queries are there overall?	Look at the hover info	1395
How much time is spent for merging state?	Look at the red area and the hover info	2487ms (78%)
How much time is spent for state consolidation?	Look at the hover info	0ms (0 times)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 times)
It looks like the execution contains many branches; how many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	14
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	14
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	369
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	746 (53% of all SMT queries)
How much time is spent in SMT solver on the longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	1292ms
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select longest path, enter `root.redraw()`, and choose `return d.durationMs;` as the histogram function in the settings popup	No significant statements are visible
What is the duration of the longest scope(beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method m_Ackermann") && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	70ms
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	7ms
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	21ms

Figure C.9: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Ackermann_function.rs.vpr

Question	Step	Answer
Does the flamegraph provide some interesting insights for the longest path?	Compare the flamegraphs obtained via `` `return t.isSmtQuery ? t.durationMs : null;` `` and `` `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` `` as the histogram function in the settings popup	There is almost no difference in shape of the flamegraphs
How does the "decisions" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.decisions");` ``	There are no significant nodes, however nodes tend to become larger to the right
How does the "decisions-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.decisions-delta");` ``	There are 6 statements with a value above 700. However, the largest value for a single SMT query is 121
How does the "propagations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.propagations-delta");` ``	There are 9 significant statements (with values above 300); 8 of them are "`execute unfold acc(…)`"; 6 of them cause "`Merge`" scopes with values over 300
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");` ``	There are 9 significant statements (with values above 100); 8 of them are "`execute unfold acc(..)`", which is very similar to above's metric
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.conflicts-delta");` ``	2 statements stand out (having values of 24 resp. 27): "`execute fold acc(isize(_22), write)`" and "`execute fold acc(isize(_21), write)`". However, their "`merge`" scopes are split up into 8 resp. 9 SMT queries of each value 3; The largest value per SMT query on longest path is 11

Figure C.10: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Ackermann_function.rs.vpr

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	“method m_Fibonacci_sequence$ $recursive_fibonacci$opensqu0closesqu $” (2336ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	1832ms (78% of the entire execution duration)
How many SMT queries are there overall?	Look at the hover info	959
How much time is spent for merging state?	Look at the red area and the hover info	1810ms (77% of the entire execution duration) 144 merges
How much time is spent for state consolidation?	Look at the hover info	0ms 0 consolidations
How much time is spent for removing permissions?	Look at the hover info	0ms 0 permission removals
It looks like the execution contains many branches; how many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	27
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	634
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	306
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	845 (88% of all queries)
How much time is spent in SMT solver on the longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	1600ms
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select longest path, enter `root.redraw()`, and choose `return d.durationMs;` as the histogram function in the settings popup	Many small nodes, no significant ones
What is the duration of the longest scope on the longest path (beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method m_borrow”) && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	71ms
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	23ms
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	23ms

Figure C.11: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Fibonacci_sequence.rs.vpr

Question	Step	Answer
Does the flamegraph reveal some interesting insights for the longest path?	Compare the flamegraphs obtained via `` `return t.isSmtQuery ? t.durationMs : null;` `` and `` `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` `` as the histogram function in the settings popup	In both flamegraphs, no significant nodes visible
How does the "decisions-delta" SMT statistics evolve on longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.decisions-delta");` ``	There are 2 statements with a value above 1000: "`execute fold acc(usize(_21), write)`" and "`execute inhale acc(usize(_26)) && acc(usize(_29) && acc(usize(_30)))`"
How does the "propagations-delta" SMT statistics evolve on longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.propagations-delta");` ``	There is one statement with a value above 1000: "`execute inhale acc(usize(_26)) && acc(usize(_29) && acc(usize(_30)))`"
How does the "quant-instantiations-delta" SMT statistics evolve on longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");` ``	There is one statement with a value above 100: "`execute inhale acc(usize(_14)) && acc(usize(_15))`"
How does the "conflicts-delta" SMT statistics evolve on longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");` ``	There are about 7 statments standing out, 2 having values above 20: "`execute exhale acc(usize(_17.val_ref.enum_0_0, read$()))`" (value 25) "`execute fold acc(usize(_21))`" (value 36)
What is the execution time of the longest path?	Enter `` `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` `` in JS console	2032ms

Figure C.12: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Fibonacci_sequence.rs.vpr

Question	Step	Answer
What is the longest member and how long does it take?	Look at the top-most scope	Method "m_Towers_of_Hanoi_spec$ $move_$opensqu0closesqu$" (1750ms)
How much time is overall spent in the SMT solver?	Look at the yellow area and the hover info	1251ms (71%)
How many SMT queries are there overall?	Look at the hover info	630
How much time is spent for merging state?	Look at the red area and the hover info	1351ms (109 merges) (77%)
How much time is spent for state consolidation?	Look at the hover info	0ms (0 consolidations)
How much time is spent for removing permissions?	Look at the hover info	0ms (0 removals)
How many branch points exist?	Enter `root.sum((n) => Math.max(0, n.successors.length - 1) + Math.max(0, n.interLevelSuccessors.length - 1)) + 1` in JS CLI	10
How many paths are there?	Enter `root.mapAfter((n, successorsResult, interLevelSuccessorsResult, subDagResult) => { if (subDagResult != null) { return subDagResult} if (successorsResult.length > 0) { return successorsResult.reduce((a,b) => a + b);} if (interLevelSuccessorsResult.length > 0) { return interLevelSuccessorsResult.reduce((a,b) => a + b);} return 1;}, false)` in JS CLI	24
How many statements are there?	Enter `root.sum((n) => n.label.startsWith("execute") ? 1 : 0)` in JS CLI	230
How many SMT queries are on the longest path?	Enter `var smtCount = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtCount++; } return n; }); smtCount` in JS console	569
How much time is spent in SMT solver on the longest path?	Enter `var smtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery) { smtDuration+=n.durationMs; } return n; }); smtDuration` in JS conosle	1136ms (90% of the total SMT solver time)
What is the distribution of the durations on the longest path?	Enter `root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { n.weight = 1; return n; });` to select the longest path, enter `root.redraw()`, and choose `return d.durationMs;` as the histogram function in the settings popup	There are 3 statements taking longer than 60ms: - "execute unfold acc(i32(_10.tuple_0))" (63ms) - "execute inhale acc(i32(_8)) && acc(i32(_11)) && acc(i32(_12)) && acc(i32(_13))" (79ms) - "execute inhale acc(i32(_18)) && acc(i32(_21)) && acc(i32(_22)) && acc(i32(_23))" (77ms) While the latter two statements include many small SMT queries, the first one consists of a relatively long SMT query: "prover assert _6@50@01 == $t@151@01" taking 52ms
What is the duration of the longest scope on the longest path (beside the method itself)?	Enter `var maxDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (! n.label.startsWith("method m_Towers_") && n.durationMs > maxDuration) { maxDuration=n.durationMs; } return n; }); maxDuration` in JS console	79ms
What is the longest duration of an SMT query on the longest path?	Enter `var maxSmtDuration = 0; root.transformLongestPath((n) => n.getMaxPathValue((n) => [n.durationMs, true]), (n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; } return n; }); maxSmtDuration` in JS console	52ms
What is the longest duration of an SMT query in the entire method?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	52ms
What is the longest duration of an SMT query in the entire method except "prover assert _6@50@01 == $t@151@01"?	Enter `var maxSmtDuration = 0; root.eachBefore((n) => { if (n.isSmtQuery && n.label !== "prover assert _6@50@01 == $t@151@01" && n.durationMs > maxSmtDuration) { maxSmtDuration=n.durationMs; }}); maxSmtDuration` in JS console	10ms

Figure C.13: Page 1 of the profiling minutes for analyzing the symbolic execution of the Viper program Towers_of_Hanoi_spec.rs.vpr

Question	Step	Answer
Does the flamegraph reveal some interesting insights for the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.num-allocs-delta");` `` in the settings popup	The same 3 statements as before are significant
How does the "decisions-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.decisions-delta");` ``	3 statements stand out, however only 2 of them as before: - "`execute inhale acc(i32(_8)) && acc(i32(_11)) && acc(i32(_12)) && acc(i32(_13))`" (value 891) - "`execute inhale acc(i32(_15)) && acc(i32(_16))`" (value 632) - "`execute inhale acc(i32(_18)) && acc(i32(_21)) && acc(i32(_22)) && acc(i32(_23))`" (value 1355)
How does the "quant-instantiations-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics.quant-instantiations-delta");` ``	Other statements appear as significant, however there is no statement with a value above 14.
How does the "conflicts-delta" SMT statistics evolve on the longest path?	Set the histogram function to `` `return t.getValue("data.additionalData.smtStatistics. conflicts-delta");` ``	The flamegraph's shape looks totally different and the sum of "conflicts-delta" on the longest path make up only a fraction of the sum of "conflicts-delta" of the entire method. Furthermore on the longest path, a different significant method shows up (as observed so far)
What is the execution time of the longest path?	Enter `` `root.subDag.getMaxPathValue((n) => [n.durationMs, true])` `` in JS console	1592ms

Figure C.14: Page 2 of the profiling minutes for analyzing the symbolic execution of the Viper program Towers_of_Hanoi_spec.rs.vpr

Bibliography

[1] Linard Arquint. D3-DAG: Arquint. https://observablehq.com/@arquintl/d3-dag-arquint. [Online; accessed 13-September-2019].

[2] Linard Arquint. Layout Algorithm for Rectangles. https://github.com/erikbrinkman/d3-dag/pull/22. [Online; accessed 13-September-2019].

[3] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust Types for Modular Specification and Verification. Technical report, ETH Zurich, 2019.

[4] Alessio Aurecchia. Visual Debugging for Symbolic Execution, 2018.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[6] Nils Becker, Peter Müller, and Alexander J. Summers. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing.

[7] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007.

[8] James Bornholt and Emina Torlak. Finding Code That Explodes Under Symbolic Evaluation. *Proc. ACM Program. Lang.*, 2(OOPSLA):149:1–149:26, October 2018.

[9] Mike Bostock. Data-Driven Documents. https://d3js.org. [Online; accessed 13-September-2019].

[10] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, RFC Editor, December 2017.

[11] Erik Brinkman. d3-dag. https://github.com/erikbrinkman/d3-dag. [Online; accessed 13-September-2019].

[12] Paul C. Bryan, Kris Zyp, and Mark Nottingham. JavaScript Object Notation (JSON) Pointer. RFC 6901, RFC Editor, April 2013.

[13] Andreas Buob. Recording Symbolic Execution, 2015.

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[15] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[16] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[17] Catapult-Project. Catapult Tracing ReadMe. https://github.com/catapult-project/catapult/blob/master/tracing/README.md. [Online; accessed 04-September-2019].

[18] Ivo Colombo. Debugging Symbolic Execution, 2012.

[19] Leonardo de Moura. Which statistics indicate an efficient run of Z3? https://stackoverflow.com/a/6847467/1990080. [Online; accessed 09-September-2019].

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Docker. Docker Hub. https://hub.docker.com. [Online; accessed 01-October-2019].

[22] Docker. What is a Container? https://www.docker.com/resources/what-container. [Online; accessed 01-October-2019].

[23] Jonas Felber. Precise And Scalable Fund Tracking On Ethereum, 2019.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[25] Programming Methodology Group. Download Viper. https://www.pm.inf.ethz.ch/research/viper/downloads.html. [Online; accessed 25-September-2019].

[26] Programming Methodology Group. Viper Tutorial. http://viper.ethz.ch/tutorial. [Online; accessed 30-September-2019].

[27] Martin Hentschel, Richard Bubel, and Reiner Hähnle. The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, 21(5):485–513, Oct 2019.

[28] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[29] Moritz Knüsel. Optimization of a Symbolic-Execution-Based Program Verifier, 2019.

[30] K. Rustan M. Leino. This is Boogie 2. June 2008.

[31] Microsoft. Visual Studio Code. https://code.visualstudio.com. [Online; accessed 25-September-2019].

[32] 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 - Volume 9583*, VMCAI 2016, pages 41–62, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[33] Matthew Parkinson and Gavin Bierman. Separation Logic and Abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM.

[34] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

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

[36] spray. spray-json. https://github.com/spray/spray-json. [Online; accessed 18-September-2019].

[37] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.

[38] Tricentis. Software Fail Watch: 5th Edition. Technical report, Tricentis GmbH, Vienna, Austria, 2017.

[39] Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM.

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):

Profiling Symbolic Execution

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

Name(s):	**First name(s):**
Arquint	Linard

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
Zürich, October 1, 2019

Signature(s)
Linard Arquint

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