GUIDED RANDOMIZED SEARCH OVER PROGRAMS FOR SYNTHESIS AND PROGRAM OPTIMIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stefan Heule
June 2018

Stefan Heule

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(John C. Mitchell)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Manu Sridharan)

Approved for the Stanford University Committee on Graduate Studies

---

Abstract

The ability to automatically reason about programs and extract useful information from them is very important and has received a lot of attention from both the academic community as well as practitioners in industry. Scaling such program analyses to real system is a significant challenge, as real systems tend to be very large, very complex, and often at least part of the system is not available for analysis. A common solution to this problem is to manually write models for the parts of the system that are not analyzable. However, writing these models is both challenging and time consuming. Instead, we propose the use of guided randomized search to find models automatically, and we show how this idea can be applied in three diverse contexts.

First, we show how we can use guided randomized search to automatically find models for opaque code, a common problem in program analysis. Opaque code is code that is executable but whose source code is unavailable or difficult to process. We present a technique to first observe the opaque code by collecting partial program traces and then automatically synthesize a model. We demonstrate our method by learning models for a collection of array-manipulating routines.

Second, we tackle automatically learning a formal specification for the x86-64 instruction set. Many software analysis and verification tools depend, either explicitly or implicitly, on correct modeling of the semantics of x86-64 instructions. However, formal semantics for the x86-64 ISA are difficult to obtain and often written manually with great effort. Instead, we show how to automatically synthesize formal semantics for 1,795 instruction variants of x86-64. Crucial to our success is a new technique, stratified synthesis, that allows us to scale to longer programs. We evaluate the specification we learned and find that it contains no errors, unlike all manually written specifications we compare against.

Third, we consider the problem of program optimization on recent CPU architectures. These modern architectures are incredibly complex and make it difficult to statically determine the performance of a program. Using guided randomized search with a new cost function we are able to outperform the previous state-of-the-art on several metrics, sometimes by a wide margin.

Acknowledgments

During my PhD I received support from countless individuals. First and foremost my thanks go to Alex Aiken, my advisor throughout my PhD. I would also like to thank everyone else who contributed in small or big parts, my collaborators, office mates, friends, family, administrators, support staff, and school officials. Particularly I'd like to mention, in no specific order, Manu Sridharan, Miles Davis, Rahul Sharma, Emina Torlak, Konstantin Weitz, Zachary Tatlock, Sean Treichler, Dan Boneh, Todd Warszawski, Wonchan Lee, Michael Bauer, Eric Schkufza, Clark Barrett, Brigitte Heule, Ruth Harris, Daniel Heule, Zhihao Jia, Steven Lyubomirsky, Osbert Bastani, Peter Heule, Pratiksha Thaker, Jason Koenig, Ben Parks, K. Rustan M. Leino, Jay Subramanian, Juan Jose Alonso, Satish Chandra, Xinyuan Zhang, Christine Fiksdal, Berkeley R. Churchill, Colleen Rhoades, Brandon Holt, Alessandra Peter, Brian Roberts, Michael D Ernst, John C. Mitchell, Lázaro Clapp, Manolis Papadakis, Jam Kiattinant, Lynda Harris, Patrice Godefroid, and everyone I forgot.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ability to automatically reason about programs and extract useful information from them is very important and has received a lot of attention from both the academic community as well as practitioners in industry. People have worked on a very wide range of topics in this area, for instance, race condition detection (e.g., [SVE+11, FF00, FF01, EA03, DRVK14, FF09]), formal equivalence checkers (e.g., [KPKG02, BBDEL96, PBG05, Ler06, CSBA17, SSCA13]), memory leak detection (e.g., [PH13, JY08, XA05, CPR07, HL03]), symbolic execution (e.g., [Kin76, CBM89, KPV03, APV07, BEL75, CS13, SAH+10, CDE+08]), lock usage checkers (e.g., [RLN98, FF04, BHJM07]), superoptimizers (e.g., [Mas87, BA06, SSA13, PTBD16, TSTL09, JNR02, CSBA17, JY17, BTGC16, TB13]), and countless others. These systems can provide a lot of value, by formally showing the absence of certain classes of bugs, helping debugging of erroneous code, optimizing programs, or pointing to potentially buggy code. Initially, these systems are often described and formalized on a small toy programming language that has most of the features of a real programming language, but is easier to work with. However, if these analyses and tools are to be applied to real programs and real systems, there are usually significant additional challenges and it is typically not possible to actually analyze the system in it's entirety. The problem is that real systems are (1) very large, (2) very complex, and (3) often at least part of the system is not available for analysis. To make this discussion more concrete, we consider three examples and illustrate these problems more thoroughly.

First, consider a memory leak detector for JavaScript. A memory leak in a garbage collected language like JavaScript is an object that is no longer used, but is still reachable and thus cannot be automatically collected by the garbage collector. If such leaked objects accumulate, the performance of the program can be significantly affected, or it can even stop working altogether if there is no memory left. For memory leak program analyses to be applicable to real programs, they must be able to understand the entirety of the program being analyzed. However, often real programs contain what we call *opaque code*: code that is executable, but not amenable to the program analysis. For instance opaque code arises if a library used by the program is written in a different programming

language not understood by the program analysis. In JavaScript this situation is in fact common, as even the standard library is not written in JavaScript, but natively instead, and thus inaccessible to any JavaScript program analysis. To make the leak detector useful, the analysis must understand this opaque code.

Second, we illustrate the challenges in building a formal equivalence checker for x86-64 programs. Proving formal equivalence between two programs in general is impossible, as the task is an example of an undecidable problem. However, there has been made a lot of progress on checkers that can prove or disprove the equivalence on a large class of useful examples. To build such an equivalence checker for x86-64 requires at a minimum a formal understanding of all x86-64 instructions. However, a formal description of these instructions is not readily available, and in fact it's impossible to analyze the CPU to see what an instruction does, as its internals are not accessible. Again, to build the tool we want we require a formal understanding of a part of the system that is not readily available.

Lastly, consider the example of building a superoptimizer for x86-64 programs. The tool is given a program, and it's task is to find the fastest implementation with the same behavior. In order to do that, a superoptimizer searches through the space of all programs in some way, and critically relies on being able to estimate the performance of the programs considered. However, again, x86-64 CPUs cannot be inspected directly, and thus some other way of estimating performance is required.

In all of these examples for the analysis to scale to real programs, or even to work at all, requires a formal understanding of a particular aspect of the system under study, which cannot directly be inspected. A common solution to this problem is to write a *model*, that describes the required aspect. Then, the analysis can use the model in place of the real system and perform its task. However, writing these models is both challenging and time consuming. Especially as real systems grow in complexity, the need for models increases as well, and manually writing them does not scale well.

Instead of requiring such models to be written manually, we propose the use of guided randomized search to find models automatically. We make use of the fact that models do not need to describe the system being analyzed in its entirety, but instead just need to model certain aspects with high enough fidelity. For instance, for an x86-64 equivalence checker, the required model needs to define the input-output behavior of all instructions. For an x86-64 superoptimizer, however, the model is not concerned with the functional behavior, and instead only needs to describe the performance behavior of x86-64 programs. Restricting models to a few features relevant to a specific analysis problem makes it tractable to learn models automatically even for very complex systems. Secondly, we propose the use of guided randomized search, where we start with a potentially very crude initial model. Then, this model is randomly perturbed, and compared to the previous model using a cost function that evaluates how well the new proposed model matches the real system. If the new model is a better fit, it replaces the current one, and otherwise the proposal is usually rejected and another random change is proposed. In this way, the cost function guides the search until a sufficiently accurate model is found.

Our thesis is that guided randomized search is well-suited to finding models that describe important parts of the behavior of real systems in a number of diverse contexts. In particular, we show how we can apply this core idea of using guided randomized search to find models in the three contexts used as examples above. Our contributions are as follows:

- We identify the problem of opaque code for JavaScript, and show how we can automatically learn models for opaque code using guided randomized search. When evaluated on real opaque code, we show that we can handle a large range of challenging opaque functions, and when compared with manually written models, we find that our learned models are more complete and more correct.

- We tackle the problem of finding a specification for the x86-64 instruction set. We propose a novel strategy we call *stratified search* to do so, and are able to learn significantly more formulas than any other automatic strategy. We evaluate these formulas for correctness and usability; in doing so, we identify several important errors in pre-existing formalizations.

- We consider the problem of superoptimization for x86-64 using guided randomized search, and introduce a new cost function that predicts the runtime of x86-64 programs much more accurately than previous work. We show that our approach can find faster programs than previous work, sometimes significantly so.

The work described in this dissertation was a collaboration with Alex Aiken, Satish Chandra, Eric Schkufza, Rahul Sharma, and Manu Sridharan, and some of it has been published in conference papers [HSC15, HSSA16].

Chapter 2

Background

A common theme of the contributions described in this dissertation is the use of guided randomized search inspired by the Metropolis-Hastings algorithm, a Markov chain Monte Carlo (MCMC) method. In this section we describe the algorithm that was introduced in [MRR+53] and later generalized in [Has70]. We also show how it can be used as an optimization method for highly irregular search spaces, and point to some alternative approaches that could possibly be used instead of Metropolis-Hastings.

2.1 Metropolis-Hastings Algorithm

The Metropolis–Hastings algorithm is a Markov chain Monte Carlo method for approximating a distribution $P(x)$ from which direct sampling is difficult. The algorithm proceeds iteratively, starting with some initial sample x_0, and then generates a sequence $x_0, x_1, x_2, \ldots$. As more and more samples are drawn, the sequence approximates the target distribution $P(x)$ more and more closely. Formally, the Metropolis-Hastings algorithm is shown in Algorithm 1. At any point t, to generate the next sample x_{t+1}, the algorithm generates a candidate x' by sampling from a *proposal distribution* $q(x'|x_t)$. This candidate is accepted as the next element of the sequence according to the acceptance probability α. Otherwise, the candidate is rejected and we keep $x_{t+1} = x_t$.

Asymptotically, the Metropolis-Hastings algorithm approaches a unique stationary distribution that is equivalent to $P(x)$ if the Markov process must be *ergodic*:

1. Every state must be aperiodic: the system does not return to the same state at fixed intervals.
2. Every state must be positively recurrent: the expected number of states for returning to the same state is finite.

Often, the proposal distribution is chose to be symmetric, i.e., such that $q(x|x') = q(x'|x)$. In

Algorithm 1 Metropolis-Hastings algorithm.

1: Initialize x_0 randomly.
2: **for** $i = 1, 2, 3, \ldots$ **do**
3: Compute candidate: $x' \sim q(x'|x_{i-1})$
4: Compute acceptance probability:
5: $\alpha = \min\left(1, \frac{P(x')}{P(x_{i-1})} \frac{q(x_{i-1}|x')}{q(x'|x_{i-1})}\right)$
6: Sample u uniformly at random from $[0, 1]$
7: **if** $u \leq \alpha$ **then**
8: $x_i = x'$
9: **else**
10: $x_i = x_{i-1}$

this case, the acceptance probability is simply

$$\alpha = \min\left(1, \frac{P(x')}{P(x_{i-1})}\right)$$

In this case, the proposal x' is always accepted if x' is more probable and is usually rejected otherwise. However, sometimes even a less probable x' is accepted, with a probability that is decreasing as the drop in probability for x' over x_{i-1} gets larger. For this reason, the algorithm tends to stay in high-probability regions of P, which is intuitively why this algorithms works.

2.1.1 Metropolis-Hastings for Optimization

The Metropolis-Hastings algorithm can also be used for optimization problems where the function f to be optimized is highly irregular, and thus more traditional approaches such as convex optimization methods cannot be applied. To this end, the function f can be transformed into a probability distribution, such that areas where f's value is low correspond to areas of high probability, and are thus explored more. In the limit, the global minimum of f is sampled from most often. A common approach to transform a a non-negative function f into a cost function is (as described in [GRS95])

$$P(x) = \frac{1}{Z} \exp\left(-\beta \cdot f(x)\right)$$

In general, computing the normalization constant Z is difficult, but the Metropolis-Hasting algorithm only requires ratios of two probabilities, and thus it is not required to know Z.

2.2 Alternative Randomized Search Algorithms

There are countless algorithms that can solve optimization problems like the ones we deal with in this dissertation, far too many to summarize here. However, we want to point to a few alternatives to using Metropolis-Hastings to search through the space of programs. These alternatives are interesting

in their own right, and might be applicable to the problems we present here. However, we have not used them and leave exploring this topic as future work.

Metropolis-Hastings, simulated annealing [KGV83] and even pure random search all consider a single point in the search space at a given time, and randomly modify that proposal to arrive at the next step (with different strategies for when to accept a new proposal). There is a separate group of algorithms called *population-based methods*, where a population of points in the search space is maintained over time. In particular, genetic algorithms embody this idea by using inspiration from natural selection. In every time-step, a population of points in the search space is maintained, and a new generation is produced by random mutation of individual points like before, but also by crossover between two or more points from the previous generation. The details of how this is done can vary a lot depending on the domain and exact algorithm. Holland et al. [HG89] provide a good introduction to the topic of genetic algorithms for search and optimization. Specifically when searching over programs (which is the topic of this dissertation), genetic algorithms have for instance been used to automatically generate patches [FNWLG09], produce variants of buffer overflow attacks [KHZH06], or find good hash functions [EHCRI06].

Unfortunately to the best of our knowledge there is no direct comparison of MCMC style algorithms and population-based methods like genetic program to optimize a cost function over the space of programs. One advantage of MCMC style algorithms is their simplicity to implement, as they only require a cost function and a proposal distribution. For genetic algorithms on the other hand, one also has to define crossover operators, which in the case of programs is not straight-forward. Livnat and Papadimitriou [LP16] argue that genetic algorithms are more difficult to get working well, and while they tend to find populations with a larger average valuation, they might not find the absolute best valuation.

Chapter 3

Computing Models for Opaque Code

3.1 Overview

In this chapter we consider the problem of computing *models* from *opaque* code. By *opaque*, we mean code that is executable but whose source is unavailable or otherwise difficult to process. Consider the case of a JavaScript runtime, which provides "native" implementation of a large number of utility functions, such as array functions. Natively-implemented methods are opaque to an analysis tool built to analyze JavaScript sources, and are a hindrance to effective analysis because the tool must make either unsound or overly conservative assumptions on what those native methods might do. This same situation arises for many languages and runtimes.

Opacity also is a concern when a third-party library is distributed in deliberately obfuscated form that hinders static analysis. In JavaScript, for example, an obfuscator might replace access to fields by computed names, e.g., transforming `y = x.foo` to `p = ["o","fo"]; y = x[p[1]+p[0]]`. Such changes can foil a static analyzer's ability to reason precisely about data flow through the heap.

Models provide an alternate, easy to analyze representation of the opaque code. Sometimes, models can be written by hand; however, this is tedious and error prone, and requires understanding of how the opaque code works. A need for better, automated ways of creating models has been argued in the literature [LDW03, SAP+11, JMM11, MLF13].

In this chapter, we show a new, automatic way of creating models for opaque functions, using a search-based program synthesis strategy. We observe that the behavior of an opaque function can often be indirectly observed by intercepting accesses to memory shared between the client and the opaque code. (Here, "shared" is in the sense of an object that is passed as a parameter to a called function and has nothing to do with concurrency.) In most common dynamic languages (e.g., JavaScript, Lua, Python, Ruby), interception of accesses to shared objects can be achieved using indirection via proxy objects (discussed in more detail in Section 3.2.8). We show that, surprisingly, it is in fact possible to generate useful models of opaque functions *based solely on observation of*

these shared memory accesses.

Given a set of inputs for an opaque function, our technique collects traces of (shared) memory accesses to those inputs by running the function against those inputs and recording the intercepted accesses. It then carries out a guided random search to synthesize from scratch a function whose execution results in the same sequence of reads and writes. Our strategy is a "generate-and-test" strategy, leveraging efficient native execution—simply running it—to test the quality of candidate models. Comparison of quality of models is done using a carefully designed cost function that takes into account the degree to which a model matches the available traces. Native execution lets the technique run tens of thousands of trials per minute, and it yields models that are in fact concrete code. Thus, our models are agnostic to whatever abstraction a program analysis wishes to employ. Figure 3.2 shows the model that our approach recovers for the JavaScript `Array.prototype.shift` method. Notice that the model includes complex control-flow constructs in addition to the elementary statements.

Related approaches The problem of generating an implementation from a trace of actions is not new, but to the best of our knowledge, our technique is new. Closely related work [LDW03, GHS12] in this area has used the concept of a *version-space algebra* to search through a space of candidate programs. In version-space algebra, a domain-specific data structure encodes all programs that are consistent with the traces seen so far. When a new trace is presented, the data structure is adjusted to eliminate those programs that would be inconsistent with this new trace. The final space can be ranked using domain-specific insights. The success of the version-space technique depends on careful design both of the space of domain-specific programs and of the data structure that represents them, in such a way that the knowledge embedded in a new trace can be factored in efficiently [GHS12].

While the version-space algebra approach has been very successful in specific domains such as spreadsheet manipulation, its success has not been shown on general-purpose programs. The work by Lau et al. [LDW03] handles only the situation in which the complete trace with the state of *all* variables is given at each step, along with the knowledge of the program counter; this simplifies the search space considerably. Given our execution traces, which contain only shared memory accesses and no program counter information, there is not clear way to represent all consistent programs in a compact data structure that can be efficiently updated for new traces.

Other recent work uses an "oracle-guided" approach to model generation, based on constraint generation and SMT solvers [QSQ+12, JGST10]. The generated constraints allow for a model to be constructed from pre-defined "building blocks" or "components," and they ensure the model is consistent with input-output examples obtained from running the opaque code. Here, the limitation is that the set of building blocks must be chosen very carefully to make the constraint problem tractable [QSQ+12]—it is unclear if such an approach could scale to generate models as complex as that of Figure 3.2 without excessive hand-tuning.

The alternative, then, is to adopt a generate-and-test strategy, and the important question is

how to organize the generation of candidates so as to not have to evaluate enormous spaces (even if bounded) of programs exhaustively. In this regard, recent work [FNWLG09] has shown the promise of genetic algorithms to synthesize patches to fix buggy programs. In that work, the operations of mutation and crossover are applied to an existing defective program that has both passing and failing test inputs. Mutation is applied by grafting existing code exercised in failing inputs, and this contains the search space to manageable size. Since our problem is to synthesize code from just the execution traces, we cannot readily adopt this method: there is no place to graft from. Nevertheless, this work has strongly inspired our own work in its use of statistical techniques to explore an enormous search space of code fragments.

For our setting, an approach inspired by MCMC sampling seems to be an effective search strategy, and this is borne out by our results.

Overview of Results We have implemented our approach in a tool called Mimic for JavaScript. It collects traces by wrapping function parameters with JavaScript proxy objects (see Section 3.2.8). We have found Mimic to be surprisingly effective in computing models of several of the array-manipulating routines in the JavaScript runtime. In fact, the models generated by Mimic have occasionally been of higher quality than the hand-written models available with WALA, a production-quality program analysis system [WAL]. When it succeeded, Mimic required, on average, roughly 60 seconds per method, evaluating hundreds of thousands of candidates in the process. The technique is easily parallelized, and in our evaluation we leveraged a 28-core machine for improved performance. Mimic was just as capable in extracting models from obfuscated versions of the same library methods written in JavaScript. Our implementation is available at `https://github.com/Samsung/mimic`.

As with any search-based technique, there are limitations (see Section 3.3.3 for a detailed discussion). Our approach is currently limited to discovering models of functions that primarily perform data flow, with relatively simple logical computation. We cannot generate a model for a complex mathematical function (e.g., sine), since our trace of shared memory accesses does not expose the complex calculations within the function. Nevertheless, we believe our approach can still be usefully applied in a variety of scenarios.

3.2 Approach

3.2.1 High-Level Overview

We start with an overview of our synthesis technique, using an example function from the JavaScript standard library.

Consider JavaScript's built-in `Array.prototype.shift` function. For a non-empty array, `shift` removes its first element e (at index 0), shifts the remaining values down by one index (so the value at index 1 is shifted to 0, 2 to 1, etc.), and finally returns e, e.g.:

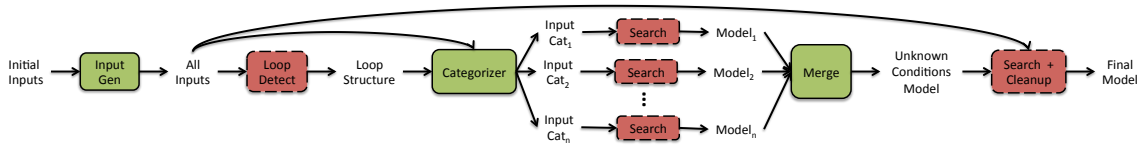

Figure 3.1: Overview of MIMIC phases. The green phases (with solid outlines) are deterministic, while the red phases (with dashed outlines) incorporate randomness. We call the procedure corresponding to the whole figure MIMICCORE.

```
function shift(arg0) {
  var n0 = arg0.length
  if (n0) {
    var n1 = arg0[0]
    for (var i = 0; i < (n0-1); i += 1) {
      var n2 = (i+1) in arg0
      if (n2) {
        var n3 = arg0[i+1]
        arg0[i] = n3
      } else {
        delete arg0[i]
      }
    }
    delete arg0[i]
    arg0.length = i
    return n1
  } else {
    arg0.length = 0
  }
}
```

Figure 3.2: Model of `Array.prototype.shift` generated by our tool.

```
var arr = ['a','b','c','d'];
var x = arr.shift();
// x is 'a', and arr is now ['b','c','d']
```

For an empty array, `shift` returns the value `undefined`. MIMIC is able to generate a model for `shift`, shown in Figure 3.2, that captures the above behaviors based solely on collected execution traces, with no access to a source representation. Note that the method also works for so-called sparse arrays, which may have certain elements missing.

In the remainder of Section 3.2, we outline the steps taken by MIMIC to generate this model. These steps are illustrated in Figure 3.1.

Generating inputs and traces Our approach begins by generating a set of execution traces for the function in question, based on some initial inputs provided by the user. An execution trace includes information about the memory operations performed by the function on any objects

reachable from its parameters, and also what value was returned. MIMIC uses proxy objects to collect such traces for JavaScript functions (see Section 3.2.8). For the `shift` function, given the input `['a','b','c','d']`, we obtain the following trace:

```
read field 'length' of arg0 //  4
read field 0 of arg0 // 'a'
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

The trace contains reads from and writes to the array parameter `arg0`, with writes showing what value was written but not *how* the value was computed.

Given traces based on the initial inputs, MIMIC generates other potentially interesting inputs whose traces may clarify the behavior of the function. In the above trace, we see entry **`read field`** `1` **`of`** `arg0`, which reads `'b'` from the input array, and then **`write`** `'b'`**`to field`** `0` **`of`** `arg0`, which writes `'b'`. Based solely on these trace entries, one cannot tell if `'b'` is being copied from the input array, or whether the write obtains `'b'` from some other computation. Hence, MIMIC generates an input with a different value in `arg0[1]`, to attempt to distinguish these two cases. Input generation is based on heuristics, and is discussed more fully in Section 3.2.4.

Loops Given a completed set of traces, our technique next tries to discover possible looping structures in the code. We first abstract each trace to a *trace skeleton*, which contains the operations performed by a trace but elides specific values and field offsets. A trace skeleton exposes the structure of the computation in a manner less dependent on the particular input values of a trace. For the above trace, the skeleton of the first four entries is **`read;read;read;write;`**.

Given a set of trace skeletons, MIMIC then proposes loop structures by searching for repeated patterns in the skeletons, rating loop structures by how well they match (see Section 3.2.5 for details). Each loop structure is assigned a probability proportional to its rating, and then MIMIC randomly selects a structure to use based on these probabilities. The full space of loop structures supported by MIMIC is covered by runs of the tool with different random seeds.

For `shift`, the highest-rated loop structure is also the correct one, and is as follows (shown over the skeleton in a regular language-like syntax):

```
read;read;(has;(read;write;|delete;))*delete;write;
```

The trace event **has** is a check if a given index is part of the array (which may return false for sparse arrays).

Categorization After a loop structure is chosen, MIMIC groups the traces into distinct *categories* based on their skeletons. For `shift`, two categories are created, one for traces matching the loop structure above, and one for the trace corresponding to the empty array (which does not match the structure). MIMIC synthesizes models for each category separately, as described next, and then merges them to discover the final model which works for all inputs.

Search To begin a random search for a model, we first generate a program to closely match one of the given input traces. For the trace above and the given loop structure, we generate the following initial program:

```
var n0 = arg0.length
var n1 = arg0[0]
for (var i = 0; i < 0; i += 1) {
  var n2 = 1 in arg0
  if (true) {
    var n3 = arg0[1]
    arg0[0] = 'b'
  } else {
    delete arg0[2]
  }
}
delete arg0[3]
arg0.length = 3
return 'a'
```

This program is clearly too specific to the given trace (it uses specific values from its input array), and its loop and if conditions are incorrect.[1]

A random mutation is then applied to this program. For instance, the second line might be replaced with the statement **var** `n1 = arg0[n0+1]`. The mutations allow the program to be gradually transformed towards a solution. To this end, the mutated program is compared with the current program in terms of how closely is matches the available traces. If it "ranks" better—as per a cost function—the mutated program becomes the current candidate. If not (like in this case, where the mutation is worse), it still becomes the current candidate with a certain probability. This is a random search inspired by the Metropolis-Hastings algorithm as described in Section 2.1.1. However, the proposal distribution used here is not symmetric, so the simplification from Section 2.1 is not

[1] In our implementation, the initial program is actually slightly more general and contains an auxiliary result variable as well as other ways to break out of the loop; see Section 3.2.2.

```
s  ::=  v := e [ e ]  |  e [ e ] := e  |  v := e ( e, ..., e )
     |  if ( e ) { s } else { s }  |  v := e
     |  for ( s ; e ; s ) { s }  |  s; s
e  ::=  ... | -1 | 0 | 1 | ... | true | false
     |  e + e  |  e - e  |  e < e  |  e == e  |  ! e
     |  ( v , ..., v ) => e
```

Figure 3.3: Syntax of a small object-oriented language.

valid and the theoretical convergence guarantees of Metropolis-Hastings might not hold. But as we will show, we find that the algorithm still works well in practice.

Our random search is able to gradually evolve this program into the code in lines 4–16 of Figure 3.2, which works for all non-empty arrays. Key to the efficacy of the search is a cost function for programs that rewards incremental progress toward a correct solution.

For the category corresponding to empty arrays, the search discovers the following program:

```
var n0 = arg0.length
arg0.length = 0
// program implicitly returns undefined
```

Merging After generating models for all categories, MIMIC merges the models using conditional statements to create a single model. At first, the branching conditions are unknown, so some trivial condition such as `true` is used. Another phase of random search discovers the correct conditions to make the model work for all input traces. Finally, a last cleanup search makes the program more readable by removing unnecessary statements and simplifying the program. For `shift`, this final search yields the model in Figure 3.2.

3.2.2 Details

Next, we detail the different phases of our approach to synthesizing models of opaque code, shown in Figure 3.1. As input, we require a function and one or more inputs to that function. Furthermore, we assume that we have a means of *executing* the function on an input of our choice and *recording a trace* of this execution. We talk more about how these assumptions are fulfilled for JavaScript in the setting of MIMIC in Section 3.2.8.

We explain our approach for a simple, dynamic, object-oriented language with object allocation, field reads and writes, functions ("callable" objects), integers, arithmetic operations, conditionals, and loops. For simplicity, all field names are integers, and objects can be viewed as arrays, where the ith array element is stored in field $i - 1$. The syntax for statements and expressions is given in Figure 3.3.

A program trace captures the execution of a function on a particular input and is a sequence of

trace *events* and can be field reads, writes and function invocations. Finally, the trace contains a return value. For all the values in the trace (such as the field being read, or the receiver), the trace directly holds that value if it is a primitive (i.e., an integer). If the value is an object, then the trace holds a unique identifier for that object. For our simple language, this trace format is sufficient to capture all object accesses performed by opaque code to objects passed as parameters. Note that the trace does not contain information about where values originate.

3.2.3 Main Loop

Algorithm 2 Main loop for model synthesis

Input: opaque function f, initial inputs I, timeout t

1: **procedure** FINDMODEL(f, I, t)
2: $m \leftarrow null$
3: **while** $m = null$ **do**
4: INITRANDOMSEED()
5: $m \leftarrow$ MIMICCORE(f, I) run with timeout t
6: **return** m

Algorithm 2 gives pseudocode for the main loop of our technique. In each iteration, a global pseudo-random number generator is initialized with a fresh random seed, and then the synthesis procedure MIMICCORE (shown in Figure 3.1) is invoked with some timeout t (*null* is returned if t is exceeded). Recall from Figure 3.1 that the loop detection and search phases make use of randomness. Hence, by periodically restarting the search with a fresh random seed, we ensure that (1) different loop structures are explored and (2) the core search explores different parts of the state space, breaking out of local minima. Note that the loop parallelizes trivially, by simply running multiple iterations concurrently and stopping when any iteration succeeds. In our implementation, we used an exponential backoff strategy to vary the timeout (see Section 3.3).

3.2.4 Input Generation

In order to generate an accurate opaque function model in our approach, we require a set of inputs that covers all of the possible behaviors the function can exhibit. While generating such an input set is impossible in general, we have developed heuristics that have worked well for the opaque functions we have tested thus far. Given one or more representative inputs provided by the user, we automatically generate additional inputs, with three main goals:

1. Determine the flow of values through a method. As illustrated with the example trace in Section 3.2.1, with only a trace it is impossible to know if a method copies a value from another location or computes it from scratch.
2. Find dependencies on values of the inputs. For instance, the method `Array.prototype.indexOf` in JavaScript returns the index of a value in an array (or `-1` if it doesn't exist in the array). By

changing the values in the array and what value is searched for, we can learn this dependency on the input values.

3. Expose corner cases. For many functions it is possible to increase the coverage by trying corner cases such as the empty array. For instance, this reveals that `Array.prototype.pop` returns `undefined` for the empty array.

Algorithm 3 Input generation

Input: initial inputs *init*

```
1: procedure InputGen(init)
2:   R ← init, I ← init, I' ← ∅
3:   while I ≠ ∅ do
4:     for i ∈ I do
5:       t ← GetTrace(i)
6:       L ← ExtractReadLocs(t)
7:       I' ← I' ∪ GenNewVals(i, L)
8:     I ← I' − R, R ← R ∪ I, I' ← ∅
9:   return R
```

Algorithm 3 gives pseudocode for input generation. In each iteration of the **while** loop, new inputs I' are generated from a set of inputs I as follows. For every input i in I, the opaque function is first executed on i to record a trace t. From t, we extract all memory locations L that were read, where a memory location consists of a base object and field offset. We generate new inputs by calling GenNewVals, which heuristically replaces the values of input locations in L with new values.

To decide what different values to generate for a location, we use heuristics based on the type of the original value. For integers, we randomly choose values, and include a few fixed values that are likely corner cases such as `0`, `-1`, or `1`. For objects, we sometimes replace the object with a clone, to distinguish cases involving aliased parameters. Furthermore, we can remove some fields, add additional fields, or provide an empty object.

At the outermost level, we repeat the steps above until no new inputs can be generated. In our prototype, we terminated input generation after two iterations of the **while** loop, as we found that this generated a sufficient set of inputs.

3.2.5 Loop Detection

After input generation, our technique next discovers possible control-flow structure for the function, i.e., loops and conditionals (the "Loop Detection" and "Categorizer" phases in Figure 3.1). By fixing the control flow structure for the model first, the core random search can concentrate on finding the correct expressions to complete the program. Here we describe loop detection in more detail; categorizing and merging were described in Section 3.2.1.

Given a set of execution traces like ours, discovering arbitrary looping constructs (with nesting,

complex conditionals within loops, etc.) is a highly non-trivial problem. The problem can be viewed as learning a regular language form only positive examples, where the examples are the program traces, and the regular language represents the control flow. Gold [Gol67] showed that the class of regular languages is not learnable from only positive examples. However, we have had success in discovering basic looping structures with a simple, probabilistic technique based on detecting repeated patterns that appear in multiple execution traces for a function, as described below.

Algorithm 4 Loop detection

Input: set of traces T

1: **procedure** LOOPDETECT(T)
2: $S \leftarrow$ GETSKELETONS(T), $C \leftarrow \emptyset$
3: **for** $s \in S$ **do**
4: $C \leftarrow C \cup$ LOOPCANDIDATES(s)
5: **for** $c \in C$ **do**
6: $score[c] \leftarrow$ RANK(c, S)
7: $L \leftarrow$ SORT($C, score$)
8: **return** $L[i]$ with probability $\alpha_{\text{loop}} \cdot \alpha \cdot (1-\alpha)^{i-1}$

We restrict the methods to have a single loop, with potentially one conditional statement in the body. To discover loop-like patterns of this form in traces, we first abstract the traces to *trace skeletons*, which only consist of the event kinds in the trace. Such abstraction is useful since there is often some variation in the events generated by a statement in a loop (e.g., the index being accessed by an array read). Given such a trace skeleton, we enumerate candidate control flow structures in the skeleton by simply enumerating all possible loop start points, as well as branch lengths inside the loop.[2] Consider the following trace skeleton:

```
read; read; write; call; read; write; call;
read; read; write; call; read; write; call;
```

For this example, the following control flow candidates are generated, among others (in a regular expression-like syntax):

```
(read; | write; call)*
(read; (write; call;|))*
read; (read; (write; call;|))*
```

Note that the candidates must match the full trace, otherwise the candidate could not be the control flow that generated the given trace. To avoid guessing unlikely loops, one can restrict the search to loops that have at least some number of iterations, say 3.

We repeat the procedure above across skeletons of all traces to create a complete set of loop candidates. Depending on the length and the number of traces, this set can be quite large. Our next

[2] This algorithm runs in polynomial time, but could get slow for very long traces, in which case optimizations might

step is to rank the loop candidates based on their likelihood. First, we can check how many traces can be explained by a given loop structure by matching it against all trace skeletons. Intuitively, the more traces that can be explained by a possible loop, the more likely that loop is correct. Secondly, if several loops match the same number of traces, then we further rank them by the number of statements in the control flow candidate, with fewer statements ranking higher.

Once the loops are sorted according to their rank, we choose one loop at random with which to continue the search, where the ith loop is picked with probability $\alpha_{\text{loop}} \cdot \alpha \cdot (1-\alpha)^{i-1}$ for some parameters $\alpha, \alpha_{\text{loop}} \in (0,1)$. α_{loop} controls the probability that no loop candidate is chosen. In our setting, we found $\alpha = 0.7$ and $\alpha_{\text{loop}} = 0.9$ to work well.

3.2.6 Guided Random Search

Algorithm 5 Random search for a model

Input: set of inputs I, loop structure l

procedure SEARCH(I, l)
  $T_o \leftarrow \{i \mapsto \text{GETTRACE}(i)\}$ for $i \in I$
  $m \leftarrow \text{INITIALMODEL}(T_o[i], l)$ for some $i \in I$
  $c \leftarrow \text{COST}(m, T_o)$
  while $c > 0$ **do**
    $m' \leftarrow \text{MUTATE}(m)$
    $c' \leftarrow \text{COST}(m', T_o)$
    if $c' < c$ **then**
      $m \leftarrow m'$
    else
      $m \leftarrow m'$ with probability
        $\min(1, \exp(-\beta \cdot (c' - c) - \gamma))$
  return m
procedure COST(m, T_o)
  $s \leftarrow 0$
  for $(i, t) \in T_o$ **do**
    $t' \leftarrow \text{GETMODELTRACE}(m, i)$
    $s \leftarrow s + \text{COMPARE}(t, t')$
  return s

At the heart of our technique (the "Search" boxes in Figure 3.1) is a procedure strongly inspired by the Metropolis-Hastings algorithm described in Section 2.1.1. We use a cost function that evaluates a model by comparing its execution traces to those of the underlying opaque function. Finding a model corresponds to finding a minimum for the cost function in the highly irregular and high-dimensional space of all models.

The concrete instantiation of the search in this context is given in Algorithm 5. We maintain a current candidate model m, and each iteration creates a mutated model m'. We then evaluate m'

be necessary. In the cases we have looked at, this has not been a problem.

using the cost function, and replace the current model if it has a lower cost. Otherwise, the new model might still be accepted, with a probability that is proportional to the difference between the two costs. This allows the search to recover from local minima and makes it robust against the irregular nature of the search space. More precisely, if the new model has a higher cost c' compared to the current cost c, then the it is accepted with probability

$$\min\left(1, \exp\left(-\beta \cdot (c' - c) - \gamma\right)\right)$$

where β and γ are parameters that controls how often models that don't improve the score are accepted.[3] Empirically, we found a value of 8 for both β and γ to work well in our setting.

3.2.6.1 Initial Program

We generate an initial program by exactly matching one execution trace t, respecting the given loop structure (if any). So, the initial program uses the exact concrete values from t in its statements as needed. If there is ambiguity because of aliasing (e.g., the same object is passed as two parameters), we can pick any of the choices. For statements inside a loop, where values might change with every iteration, we simply use values from the first iteration. It is then the job of the random search to generalize this very specific initial program to other inputs, and generalize statements inside loops to work for all iterations.

More precisely, a trace event `read field f of o` in t is translated to the statement `n := o[f];` for a fresh local variable `n`. The trace event `write v to field f of o` is translated to `o[f] := v;` and a call `call f with a0 .. an` yields the statement `f(a0, ..., an);`. If there is a loop, then a loop header `for (i = 0; i < 0; i++)` is generated, for a fresh local variable `i`. Initially, the loop body is never executed, and it is up to the random search to find the correct termination statement. To allow breaking out of the loop early, we add an additional statement to the end of the loop body of the form `if (false) break;`.[4]

Finally, we introduce a local variable `result` for the result of the function, which gets returned at the end. To allow incrementally building the result, we additionally add the statement `if (true) result = result;` inside the loop body as well as the `if` statement containing the `break`. Initially, these are nops, but allow the search to accumulate the result if necessary.

3.2.6.2 Cost Function

Our cost function Cost in Algorithm 5 computes the cost of a candidate model by comparing its execution traces on the inputs against T_o, the traces from the opaque function. To do so, it uses a function Compare that takes as input a trace t from the opaque function called the *reference trace*,

[3] In an approach more closely resembeling MCMC, γ would be zero. However, we found a non-zero value for γ to reduce convergence times.

[4] Alternatively, we could allow more complicated termination expression in the loop header.

and t' from the current model called the *candidate trace*, both generated from the same input. It then computes a *score* that measures how close the t' is to t. A score of zero indicates $t = t'$, while any larger value indicates a difference in the traces.

For our approach to work effectively, it is crucial that the random search can make incremental progress toward a good model. This incremental progress is enabled by the cost function reflecting partial correctness of a model in the score. To this end, COMPARE gives fine-grained partial credit, even for partially-correct individual events. For instance, if the reference trace contains the event `read field 1 of #1`, and the candidate trace contains `read field 2 of #1`, then the score is lower (better) than in the case where no read event was present at all, or where the read event also read from the wrong object.

Formally, $\textsc{Compare}(t, t')$ is defined as:

$$\frac{1}{2}\left(\sum_{\substack{\text{Event}\\ \text{kind } k}} \sum_{\substack{i<|\text{evs}(t,k)|\\ i<|\text{evs}(t',k)|}} \text{dist}(\text{evs}(t,k)[i], \text{evs}(t',k)[i])\right) \tag{3.1}$$

$$+\left(\sum_{\substack{\text{Event}\\ \text{kind } k}} ||\text{evs}(t,k)| - |\text{evs}(t',k)||\right) \tag{3.2}$$

$$+ \mathbb{1}\{\text{return-value}(t) \neq \text{return-value}(t')\} \tag{3.3}$$

We use $\text{evs}(t, k)$ to refer to all events in a trace t of kind k (e.g., all field reads) as a list, which can be indexed using the notation $\ell[i]$, and $|\cdot|$ returns the length of a list (or the absolute value, depending on context). return-value$(\cdot)$ refers to the return value of a trace, and $\text{dist}(\cdot, \cdot)$ measures the similarity of two trace events (of the same kind). It is defined as follows:

$$\text{dist}(e_1, e_2) = \frac{1}{|\text{vals}(e_1)|} \sum_{i<|\text{vals}(e_1)|} \mathbb{1}\{\text{vals}(e_1)[i] \neq \text{vals}(e_2)[i]\}$$

where vals$(\cdot)$ is a list of values for a given trace event. For a field read, this is the receiver and field name; for a field write it additionally includes the value written; and, for a function call it includes the function invoked as well as all arguments. Note that $\text{dist}(\cdot, \cdot)$ is scaled to return a value between 0 and 1.

In the formula for COMPARE, (3.1) calculates the difference for all the events that are present in both traces, while (3.2) penalizes the model for any event that is missing or added (compared to the reference trace). Because (3.1) is scaled by $\frac{1}{2}$, the search gets partial credit for just having the right number of events of a given kind in a trace, even if the wrong fields are read/written. This is useful to determine the correct loop termination conditions without necessarily having a correct loop body already. Finally, (3.3) ensures that the result is correct. Section 3.3 will show the advantage of this

fine-grained cost function over a more straightforward approach.

3.2.6.3 Random Program Mutation

Algorithm 6 Random Program Mutation

Input: current program P

```
1:  procedure MUTATE(P)
2:      P' ← P
3:      s ← RANDOMSTMT(P')
4:      switch type of s do
5:          case for loop
6:              s.upperBound ← RANDOMEXPR()
7:          case conditional statement
8:              s.condition ← RANDOMEXPR()
9:          case field read
10:             if RANDOM() then
11:                 s.receiver ← RANDOMEXPR()
12:             else
13:                 s.field ← RANDOMEXPR()
14:         case field write
15:             if RANDOM(1/3) then
16:                 s.receiver ← RANDOMEXPR()
17:             else
18:                 if RANDOM() then
19:                     s.field ← RANDOMEXPR()
20:                 else
21:                     s.value ← RANDOMEXPR()
22:         case local variable assignment
23:             s.rhs ← RANDOMEXPR()
24:         case function call
25:             i ← random argument index
26:             s.args[i] ← RANDOMEXPR()
27:         case otherwise
28:             Try again with a new random statement
29:     return P'
```

The control flow structure is fixed at this point, and the random search can concentrate on finding the correct sub-expressions for all the statements. To this end, a statement is selected at random, and modified randomly, as shown in Algorithm 6: For field reads, writes and function calls, a sub-expression is selected at random and replaced with a random new expression (we will explain shortly what kinds of expressions are generated). Similarly, for local variable assignments, the right-hand side is randomly replaced. For a `for` loop, the upper bound is replaced with a new random expression. For an `if` statement, the condition is replaced randomly. `break` statements are not modified.

Generating random expression follows the grammar of the programming language in Figure 3.3. To avoid creating very large nested expressions, the probability of producing an expression of depth d decreases exponentially with d. The set of local variables than can be generated is determined by the set of variables that is defined at the given program point. The set of constants is taken from all constants that appear in any of the traces, as well as a few heuristically useful constants (such as `0`).

These program mutations define the proposal distribution in the Metropolis-Hastings algorithm. However, this proposal distribution does not fulfill all the required criteria for the asymptotic guarantees of the Metropolis-Hastings algorithm. For instance, in Section 2.1.1 we assumed the proposal distribution to be symmetric; that is the probability for transforming a point p in the search space into a point p' is the same as the reverse transformation. In our setting where the space is the space of JavaScript programs, this is not the case. In general it is difficult to correctly normalize the probabilities when transforming programs randomly to achieve the required criteria. However, the guided randomized search even without the formal guarantee appears to do a lot of useful work, as we show in Section 3.3.

3.2.7 Cleanup

When the random search succeeds, the resulting program typically contains redundant parts. For instance, if it is not necessary to break out of a loop early, then the statement `if (false) break;` can be removed. Similarly, there might be unused local variables, or expressions that are more complicated then necessary (e.g., `1+1`). To clean up the resulting model, another random search is performed, with a slightly modified cost function as well as different random mutations: In addition to the existing program transformations, statements can now also be removed. Furthermore, we add the number of AST nodes of the program to the cost. This allows the search to improve the program by removing unused statements and simplifying expressions like `1+1` to `2`. Furthermore, the cost will never reach zero, and thus the search is stopped after a certain number of iterations have been carried out. Cleanup is not strictly necessary, as the models perform the same observable behavior, whether they are cleaned up or not. However, cleanup can make programs easier to look at by humans.

3.2.8 Implementation

We have implemented the ideas presented in Section 3.2.2 in a prototype implementation called Mimic for JavaScript. The tool is open source and available online.[5] In this section we highlight challenges and solutions particular to our setting, and discuss some implementation choices.

Trace Recording using Proxies We leverage the ECMAScript 6 proxy feature [Netb], which allows for objects with programmer-defined behavior on interactions with that object (such as field

[5] https://github.com/Samsung/mimic

reads and writes, enumeration, etc.). An object `o` can be virtualized with a handler `h` as follows:

```
var p = new Proxy(o, h);
```

Any interaction with `p` will ask the handler how to handle that interaction (and default to accessing `o` directly if the handler does not specify any action). We leverage this feature to record the traces required for our technique, by proxying all non-primitive arguments to the opaque function. This way, we can intercept any interaction the opaque function has with the arguments and record it as an event in the trace. Our handler responds to all interactions just like the underlying object would (by forwarding all calls), with the additional book-keeping to record the trace.

Newly Allocated Objects. One challenge with this approach is that newly allocated objects will only be visible to the recording infrastructure when they are returned (or otherwise used, e.g., written to a field). For instance, the function `Array.prototype.map` takes an array and a function, and applies the function to all arguments, returning a new array with all the results. When we record the trace, we see all the accesses to the array elements and the function invocations, but the field writes to the newly allocated array are not visible.

To handle such functions, we generate the relevant object allocation at the beginning of the initial model and assign it to the `result` variable. We also add a number of guarded field writes to the newly allocated object at different locations in the model (e.g., before the loop or inside the loop body): `if (false) result[0] = 0;`. The random search is then able to identify which particular assignments are correct (by changing the guard) and find the correct field name and value to be written.

Non-Terminating Models It is easy for the random mutations to generate programs that are non-terminating, or take very long to terminate. This can cause an infinite loop when recording the traces as part of evaluating the cost function. Luckily, given the reference trace, we know how long the trace from the model should be, and we can abort the execution of a model when the trace gets significantly longer than the reference trace. In that case, the cost function assigns a very large cost to the trace (note that small increases in trace length are not excessively penalized).

JavaScript-Specific Features JavaScript contains various features not in our language from Section 3.2.2. Many of these are straightforward to support, e.g., deletion of fields, function receivers, and exceptions. JavaScript allows variadic functions by providing the special variable `arguments` that can be used to access any argument using `argument[i]`, as well as the number of arguments passed with `arguments.length`. For instance, the function `Array.prototype.push` adds zero or more elements to an array. MIMIC supports such functions, by generating inputs of different lengths and generating expressions of the form `arguments[i]` and `arguments.length` during the random search. Essentially, we can view the function as just having a single argument that is an array, and

then use the usual input generation strategy described earlier. The only difference is that we do not get any information in the trace about accesses to `arguments` (this special object cannot be proxied).

Optimizations It is important for the random search to be able to evaluate many random proposals, which is why we implemented the following optimizations to our approach.

Input Selection. Input generation can create thousands of inputs (or more), and executing all of them during search would be prohibitively slow. For this reason, we restrict the inputs used during search to a much smaller number. We found 20 inputs to work well in our experiments, if the inputs are diverse enough. To get diverse inputs, we choose input that generate traces of different lengths, as well as inputs that have different scores on the initial program. If the search succeeds, all inputs are used to validate the result. If in this final validation, the score for some input is non-zero, then the search is considered to have failed, and a new run (with a different) random seed as described earlier is necessary. We found this to not be an issue.

Program Mutations. If the program mutations are generated naïvely, it is easy to generate nonsensical programs. For instance, by just following the programming language grammar, one might generate an expression that tries to read a field of an integer, or use an array object as a field offset. To avoid exploring such malformed programs, we filter out expressions when we can statically decide that they are invalid. Given the dynamic nature of JavaScript, this is not always possible, but we found that our filtering eliminates many common malformed expressions.

Parallelization Strategy. Our procedure FindModel is embarrassingly parallel. However, one can further improve performance by exploiting the fact that some of the successful runs finish much more quickly than others. It is often better to kill a search early and retry again, in the hope of finding one of those very quick runs. To do this, we start with a small initial timeout t_0, and then exponentially increase it by a factor f. We found that running 28 threads in parallel with a timeout of $t_0 = 3$ seconds to work well, with a factor of $f = 2$.[6] All threads run with the same timeout, and if any of them succeed, all others can be aborted and the model can be returned. If none succeed, then the timeout is increased by a factor of f, and the process starts again.

Cleanup As noted earlier, cleanup is not necessary, strictly speaking, as models exhibit the same observable behavior with or without cleanup. For this reason, our prototype implements a quick cleanup that only removes unnecessary statements and is always applied. Then, a full cleanup as a random search can be applied optionally, to make the models more readable.

3.3 Evaluation

Here, we present an experimental evaluation of Mimic on a suite of JavaScript array-manipulating routines. We evaluated Mimic according to the following five research questions:

[6] If run with fewer threads, Mimic will automatically use a smaller factor, so that roughly the same number of short runs are made.

(RQ3.1) Success Rate: How often was MIMIC successful in generating an accurate model for the tested routines?

(RQ3.2) Performance: When it succeeded, how long did MIMIC take to generate a model?

(RQ3.3) Usefulness: Were the models generated by MIMIC useful for a program analysis client?

(RQ3.4) Obfuscation: Is MIMIC robust to obfuscation?

(RQ3.5) Cost Function: How important was the fine-grained partial credit given by our cost function (Section 3.2.6.2)?

3.3.1 Experimental Setup

Our primary subject programs were the built-in methods for arrays on `Array.prototype` provided by any standard JavaScript runtime [Neta]. These methods exhibit a variety of behaviors, including mutation of arrays, loops that iterate forward and backward through the array, and methods returning non-trivial computed results (e.g., `reduce`). Furthermore, many of these methods can operate on JavaScript's sparse arrays (in which certain array indices may be missing), necessitating additional conditional logic.

We ran our experiments on a Intel Xeon E5-2697 machine with 28 physical cores, running at 2.6 GHz and using `node v0.12.0` to execute MIMIC (written in TypeScript and Python). Our implementation parallelizes the search for a model using different random seeds.

3.3.2 Results

Success Rate MIMIC was able to generate models for some of the `Array.prototype` functions, but not others. The functions for which it succeeded are listed in Table 3.1; we discuss the other functions below. The models we synthesized can be found at `https://github.com/Samsung/mimic/blob/master/models/array.js`.[7] We also included three other functions over arrays (`max`, `min`, and `sum`) that performed a bit more computation than the built-in functions.[8]

Performance Experimental data addressing **(RQ3.2)** appears in Table 3.1. The performance of MIMIC was quite reasonable, with models taking an overall average of 60.6 seconds to generate using our exponential increasing timeout strategy, and less than 5 minutes on average for all models. This experiment was repeated 100 times per example, without using the full cleanup (see Section 3.2.8). The additional time it would require to perform a full cleanup is 3.74 seconds on average, and less than 8 seconds for all examples (for the default of 1000 cleanup iterations).

In the same table we also show the loop rank assigned by our ranking heuristic (1 being the highest ranked proposal). The loop ranking chooses the correct control structure for 9 out of 15 examples with rank 1, and with our choices for α and α_{loop}, MIMICCORE then picks this loop with probability 64%. For the five examples where the correct loop is ranked second, the probability is

[7] Some array methods can handle "Array-like" objects; we have not used such inputs and focused only on actual arrays.

[8] Though code is available for these functions, MIMIC made no use of it, observing their behavior as if they were opaque.

Function	Time to synthesize (in seconds)	Loop rank
every	67.86 ± 22.41	1
filter	43.05 ± 16.13	1
forEach	4.59 ± 1.52	1
indexOf	42.94 ± 38.59	1
lastIndexOf	36.92 ± 19.22	2
map	15.64 ± 6.86	1
pop	2.35 ± 0.74	loop-free
push	291.94 ± 310.17	3
reduce	25.33 ± 12.51	1
reduceRight	126.41 ± 53.77	1
shift	117.54 ± 52.11	1
some	6.74 ± 3.16	1
max	56.61 ± 107.86	2
min	50.69 ± 121.95	2
sum	20.35 ± 39.83	2

Table 3.1: Summary of all `Array.prototype` functions that MIMIC can handle, as well as some handwritten functions to compute the sum, maximum and minimum of an array of numbers. We report the average time (over 100 runs) it takes to synthesize the model (using the quick cleanup) on our hardware as well as how high the correct loop was ranked (1 being the highest rank).

18.9%, and for `push` with rank 3 it is 5.7%. Finally, `pop` does not have a loop (and the loop inference does not propose any loops).

Longer search times were due to several reasons: (1) loop ranking, (2) complicated expressions (e.g., generating the index `n0-i-1` is lower probability than, say, `i`), and complex dependencies between loop iterations (`reduce` requires the result of the one iteration to be passed in the next iteration).

MIMIC currently cannot synthesize models for `Array.prototype` methods not listed in Table 3.1, due to the following issues:

- **Multiple loops**: `concat`, `sort`, `splice` and `unshift` all require multiple loops, for which we currently do not have any heuristics.
- **Complex conditionals within loops:** `reverse` reverses a list in place. The loop body takes two indices from the front and back and exchanges them. To handle sparse arrays, four different cases need to be handled (depending on whether either element is present or not).
- **Lack of relevant mutations:** `join`, `toString` and `toLocalString` require computing a (possibly localized) string representation of an arbitrary array element, which our mutations do not propose at the moment.
- **Proxy-related bugs:** We discovered some bugs in the current proxy implementation in

`node.js`. Unfortunately, it crashes for `slice`. It also reports traces that are not in accordance with the specification for `concat`, `shift` and `reverse`. For `shift` we used our own implementation that follows the specification.

Usefulness To answer **(RQ3.3)**, we compared the models generated by MIMIC with those present in the T.J. Watson Libraries for Analysis (WALA) [WAL], a production-quality static analysis framework for Java and JavaScript. We found that WALA did not yet have any model for functions `reduce`, `every`, and `some`. Since these functions perform callbacks of user-provided functions, a lack of a model could lead to incorrect call graphs for programs invoking the functions. We added the MIMIC-generated models for these functions to WALA, and confirmed via small examples that their presence improved call graph completeness. Additionally, we found WALA's existing model of the frequently-used `pop` function was written incorrectly, such that it would always return `undefined` rather than the popped value. Many of WALA's models also do not handle sparse arrays, which MIMIC models do handle. These examples illustrate how writing models by hand can be tedious and error-prone, and how tool support can ease the process. MIMIC-generated models for the above functions were accepted for inclusion in WALA.[9]

Obfuscation To answer **(RQ3.4)**, we ran MIMIC-generated models through a well-known JavaScript obfuscator,[10] and tested that MIMIC generated the same model for the obfuscated function. The obfuscator employed rewrites static property accesses into dynamic ones (see Section 3.1), which could significantly degrade the quality of many static analyses. We confirmed that for all functions, we obtained the same model when using the obfuscated code as the baseline, showing the promise of our technique for making such code more analyzeable.

Cost Function To answer **(RQ3.5)**, we ran MIMIC with a more naïve cost function that only gave credit to a trace when an individual event matched exactly with the reference trace, rather than giving partial credit for partial event matches (see Section 3.2.6.2). In an experiment, we found that our cost function led to decreased running times compared to the naïve function, validating our use of fine-grained partial credit. Specifically, the running time average increased by 39.7% on average for the naïve cost function, and as much as 170% for some examples.

3.3.3 Limitations and Threats to Validity

MIMIC currently cannot handle opaque functions with any of the following properties:

- *Complex local computations*, e.g., a sine function, as our trace does not expose such computations.
- *Nested loops or complex conditionals inside loops*, as discussed above. Adding such support without excessively increasing the number of loop candidates is future work.
- *Reliance on non-parameter inputs*, e.g., a variable in the closure state initialized via some other

[9] https://github.com/wala/WALA/pull/64
[10] http://www.javascriptobfuscator.com/Javascript-Obfuscator.aspx

sequence of method calls. This could perhaps be handled by the user providing a richer test harness for the function.

- *Global side effects*, e.g., on the network or file system. This is a limitation of the native execution approach.
- *Long running time*, which will slow down our search, again due to native execution.

The primary threat to validity of our evaluation is our choice of benchmarks. We tested MIMIC on all `Array.prototype` methods, and discussed cases it could and could not handle. But, MIMIC's handling of different array routines, or routines on other data structures, could vary.

3.4 Related Work

Summary Computation Computing summaries of procedure calls has been a very active area of in program analysis, and it is impossible to mention all the related work on the topic; Sharir and Pneuli's seminal work [SP81] is a good overview of the foundational approaches to this problem. We have used the term "models" in this chapter; summaries can be thought of as models that suffice for specific static analyses, which use abstractions.

Most of the work on summary computation assumes the availability of library code for analysis. However, recently, several authors have tried to deal with unavailable libraries. The work by Zhu et al. [ZDD13] deals with the problem of missing library code by inferring the minimal number of "must not flow" specifications that would prevent undesirable flows in the client application. Bastani et al. [BAA15] have also presented an approach that infers potential summaries for an unavailable library, based on a particular static analysis problem being solved. In both the above approaches, ultimately a human auditor must judge the validity of the findings of the tool. Like these works, we also do not require the code to be available in analyzable form, but (unlike these works) we do assume the library code is available to execute. On the other hand, while the summaries computed in the above mostly capture tainted-ness and aliasing, the models that we compute are considerably richer. Clapp et al. [CAA15] prevent work to generate explicit information flow specifications from concrete executions. Like our work, they rely on execution the opaque code, but their models are specific to just information flow specifications.

Madsen et al. [MLF13] construct models of library functions through static analysis of uses of values returned from native JavaScript functions. However, their work does not infer flow of values *through* the library functions as would be needed, for instance, for alias analysis; rather, it only infers the type structure of values returned from the libraries.

Summaries can be helpful in dynamic analysis settings as well. Palepu et al. [PXJ13] compute summaries of library functions with the intention of avoiding repeated execution of instrumented library code. Their summary representation is suitable for aliasing and taint-flow like properties. By contrast, our technique requires only partial observation of the execution of the library code, and our

models are richer, but they are more expensive to compute.

Trace-based Synthesis As mentioned in Section 3.1, there has been considerable work related to the problem of constructing programs to fit traces. One of the earlier papers in this area is that of Biermann et al. [BBP75]. Given a trace of *all* instructions, as well as conditionals, they show how to construct a (looping) program that would explain the trace. They present a backtracking procedure that tries to discover the right program structure. However, the technique requires all instructions, including conditionals, in the trace. By contrast, our technique requires observing only the shared reads and writes.

Traces can be detailed traces of low-level instructions, or they can be traces of high level domain-specific instructions, e.g. deleting a character in an editing buffer. Automatic construction of "programs" to automate end-user tasks has been an area of much work in the last decade [LDW03, GHS12]. Repetitive user actions correspond to example traces, and a tool synthesizes a program representation that automates such user actions. However, most current approaches to synthesis from examples are limited to generating programs in carefully-designed domain-specific languages, leveraging clever algorithms and data structures for efficiency (cf. version algebra discussion in Section 3.1). Extending such approaches to generating programs with richer data and control structures, as needed in our scenario, is non-trivial. When it comes to general purpose programs, Lau et al.'s work [LDW03] mentions the increased difficulty in synthesis when traces do not capture all the steps of a program's execution; in fact, they report experiments only on complete traces, as in Biermann's work.

Other Synthesis Approaches Jha et al. [JGST10] presented a technique to synthesize straight-line programs from examples. Their technique searches through a space of combinations of program operations which could be either primitive statements, or API calls. Like our work, they check the correctness of their synthesized programs against a native implementation on a set of inputs. Later, Qi et al. [QSQ+12] extended the technique to handle loops and data structures by extending the set of possible primitive statements. As discussed in Section 3.1, the main limitation of this approach is in scalability of the constraint solver as the number of primitive statement types increases; our generate-and-test approach leverages native execution and parallelization to improve scalability.

Sketching [SLTB+06] is another synthesis system based on constraint solving. The idea of sketching is to synthesize missing expressions (a.k.a. "holes") from an otherwise complete program. For certain domains, boolean satisfiability (SAT) techniques can be used to efficiently synthesize completions of programs with holes. Kuncak et al. [KMPS10] have presented automated synthesis procedures that work (based on formula solving) for a limited setting of linear arithmetic and data structures. In our setting, the entire program needs to be synthesized, and hence the above constraint-based techniques do not apply directly.

Other Uses of Search Our work is inspired by recent successes of search-based techniques in the research community. To pick two examples, there is the automatic patch generation work

by Weimer at al. [FNWLG09], which uses genetic algorithms for search (and which we discussed in Section 3.1), and compiler optimization work by Schkufza et al. [SSA13], which also uses the Markov-Chain Monte Carlo technique.

Chapter 4

Stratified Synthesis

4.1 Overview

In this chapter we turn our attention to a much lower-level programming language and consider the various x86 instruction sets that have been ubiquitous for decades and are used on the majority of all server and desktop machines. Because of the importance of x86, many software analysis tools have been built that reason about programs written in x86 directly (e.g., [CKG05, BGRT05, RB08]) or that reason about programs in high-level languages that are then translated to x86 (e.g., [Ler12]). All such tools require a precise formal semantics of the x86 ISA. One might assume, then, that a formal semantics of the current 64-bit x86 ISA (known as x86-64) would be readily available, but one would be wrong. In fact, the only published description of the x86-64 ISA is the Intel manual [Int15] with over 3,800 pages written in an ad-hoc combination of English and pseudo-code. There are two primary barriers to producing a formal specification of x86-64. First, the instruction set is huge: x86-64 contains 981 unique mnemonics and a total of 3,684 instruction variants. Second, many of the instructions have complex semantics. For example, the rotate instruction implicitly truncates its integer argument, modifies an implicit register operand, and depending on its input either changes the value of multiple condition registers or leaves them undefined. Many instructions have comparable edge cases and variations, and the behaviors are often inconsistent with other, related instructions.

For those who need a formal semantics of x86-64, a common though necessarily inefficient approach is to generate formal specifications by hand on demand: whenever progress requires the semantics of an instruction i, someone writes a formula for i. Many software research groups have taken this approach, however our personal experience is that it is nearly impossible to write and maintain correct specifications solely by hand—x86-64 is just too big and complicated. Prior to the work reported in this dissertation, we gradually built an elaborate infrastructure to support developing a formal semantics, including templates to describe the semantics of families of closely related instructions and testing tools to compare the formal descriptions with the behavior of the actual hardware. Wherever

we added automation, we found additional bugs in the human written specifications. Eventually we came to ask whether we could eliminate the problematic human effort entirely and generate a useful formal description of most of x86-64 from a small "proto specification".

The most successful previous attempt to automatically generate a formal semantics for x86-64 is a template-based method for explaining input-output examples that are derived from hardware executions. This approach has been used to generate formulas for 534 instruction variants [GT12]. These variants cover the core of 32-bit x86, but do not include any of the more recent extensions of x86-64 (e.g., vector instructions). The templates are written by hand and must satisfy certain structural restrictions. It seems unlikely that these requirements are expressive enough to cover the remainder of the instruction set.

In this chapter we demonstrate a new technique that can automatically synthesize a formal semantics for a large fraction of the x86-64 ISA. Our approach uses program synthesis to learn bit-vector formulas that can be consumed by SMT solvers such as Z3 [WHdM13] or CVC4 [BCD+11]. We begin by choosing a small set of x86-64 instructions that we call the *base set*. Our approach expects a formal semantics for every instruction in the base set as input and then generates formulas for the remaining instructions automatically. The effort required to produce the formulas for the base set is small—they can either be written by hand or generated using some other approach (including [GT12]).

At a high level, our approach works as follows. We execute an instruction i for which the formal semantics is not known yet on a set of test inputs T to obtain an initial description of its behavior. We then search for a program p that matches the behavior of i on the tests T, where p only uses instructions drawn from the base set S. However, there is little reason to have confidence that one program that happens to match an instruction's behavior on a set of test cases actually captures the behavior of that instruction for all possible inputs, so we perform the search multiple times to find a set of programs P that match the behavior of i on T and use only instructions from S. Given two programs $p, p' \in P$, we test whether $p \equiv p'$ using an SMT solver and the formulas from the base set. If the two programs are semantically distinct (meaning the agreement on T is coincidental), we know that one or both programs are not a correct description of i. We use the model produced by the SMT solver to obtain an input t that distinguishes p and p', add t to the set of tests T, and start over. We repeat this process until we are unable to generate another program not in P that agrees with i on T (which includes all counterexamples discovered along the way). When we are done, we choose one $\hat{p} \in P$ and return the formula for $\hat{p}$ as the semantics of i.

Even given that we search for multiple programs until no further inconsistencies can be discovered, there is no guarantee that the program $\hat{p}$ correctly implements i. This uncertainty is unavoidable: we do not start with a formal specification of i's behavior, and so the best we can possibly do is to produce a semantics that is consistent with whatever information is available. However, we have carefully compared our automatically generated semantics to previously developed formulas for

x86-64 and found a number of discrepancies. In every case, the automatically generated semantics was correct and the hand-written semantics was wrong. Furthermore, as a result of this effort we discovered a number of new bugs and inconsistencies in the Intel manual (see Section 3.3.2).

Because we want to automate the production of the formal semantics as much as possible, we must minimize the size of the base set of instructions that is the input to the process. However, many of the instructions that we consider can only be characterized by programs that are so long that no currently available program synthesis technique can produce them if they are written using only instructions in the initial base set. To address this problem, we introduce the idea of *stratified synthesis*: whenever we learn a program p for an instruction i, we add the formula for p to the base set as the semantics of i. Thus, as we learn the semantics of simpler instructions, our vocabulary for expressing the semantics of more complex instructions expands. We have implemented the approach in a tool called STRATA and we empirically observe that the use of stratified synthesis is both effective and necessary—there are many instructions that are learned only after other, intermediate, instructions have been learned.

4.1.1 Modeling x86-64

In this chapter, we focus on the functional aspects of the x86-64 instruction set. That is, we search for bit-vector formulas that give a precise description for the input-output behavior of instructions on registers and memory. The details of the x86-64 memory model, such as how memory behaves under concurrency or alignment requirements are important but orthogonal to that goal and are not inferred by our approach. Nonetheless, we consider the majority Haswell instruction set, with extensions such as AVX and AVX2.

4.1.2 Modeling the CPU State

We model the CPU state as bit-vectors that correspond to registers and an array that represents a byte-addressable memory. We restrict our register model to the registers described below. The x86-64 architecture has additional special purpose registers and flags used by system-level instructions, which we do not attempt to model or learn (see Section 4.1.3).

- **General Purpose Registers:** The 16 64-bit registers: **rax**, **rcx**, ..., **r15**; the lower 32, 16, and 8 bits of those registers: **eax**, **ax** and **al**, etc.; and the legacy registers that name bits 8 through 15: **ah**, **ch**, **dh**, **bh**.
- **Vector Registers:** The 16 256-bit registers: **ymm0**, ..., **ymm15**; and the lower 128 bits of those registers: **xmm0**, ..., **xmm15**.
- **Status Flags:** Five bits from the **rflags** register. These store partial results from many arithmetic operations and are used by conditional jump instructions: **cf** (carry), **pf** (parity), **zf** (zero), **sf** (sign), and **of** (overflow).

4.1.3 Instructions in Scope

We exclude a few classes of instructions because they are very difficult to model, rarely used, or both. Of the entire 3,684 instruction variants that make up the x86-64 Haswell ISA, these exclusions leave us with 2,918 variants whose specifications we can potentially infer. The excluded instructions are:

- **Systems-level** (302 variants): These instructions are rarely used by application-level code and would require a more detailed model of the operating system, protection levels, and other low-level details. Examples are `hlt` (stopping execution), `syscall` (system call) and `invpcid` (invalidate entries in the TLB).
- **Cryptography** (35 variants): These instructions support AES encryption. An example is `aeskeygenassist` (AES round key generation).
- **x87** (155 variants): These are floating-point instructions introduced in 1980 and deprecated by SSE (1999). The newer instructions are faster, more versatile, and vectorized.
- **MMX instructions** (177 variants): These are vector instructions introduced in 1997 and deprecated by SSE (1999). These instructions suffer from limitations such as the inability to interleave integer and floating-point operations.
- **String instructions** (97 variants): Currently unsupported by the synthesizer we use.

4.1.4 Dataflow Information

Our approach to synthesizing formulas requires information about which locations an instruction reads from and writes to with respect to both registers and memory. In most cases, these sets are given by an instruction's operands. However, some instructions implicitly read or write additional locations. For example, `mulq rcx` multiplies `rcx` (the explicit operand) by `rax` (implicit operand) and stores the result in `rax` and `rdx` (both also implicit). Certain x86-64 instructions may also place undefined values in register locations. Because these values are free to vary between implementations ([GT12] showed that different CPUs actually do exhibit different behavior), we require that these locations be specified as well, so as not to overfit to the CPU that we run our experiments on. We neither attempt to learn the output relation for undefined values that are placed in output registers, nor allow instructions to read from an undefined location. Although this information is already provided by the x86-64 specification in sufficient detail, we note that it would have been possible to automatically infer the majority of it as well.

4.2 Approach

In this section we describe our approach, starting with a simple example that shows the main steps, followed by an explanation of the details. The full algorithm in pseudocode is shown in Algorithm 7

Algorithm 7 Main algorithm

Input: base set *baseset*, a set of instruction/formula pairs, and a set *worklist* of instructions
Result: a set of instruction/formula pairs

```
 1: procedure STRATA(worklist, baseset)
 2:     tests ← GENERATETESTCASES()
 3:     while |worklist| > 0 do
 4:         instr ← CHOOSE(worklist)
 5:         worklist ← worklist − {instr}
 6:         t ← STRATAONE(instr, baseset, tests)
 7:         if t is failure then
                # we couldn't learn a program for instr yet
 8:             worklist ← worklist ∪ {instr}
 9:         else
10:             baseset ← baseset ∪ {⟨instr, t.formula⟩}
11:     result ← baseset
12:     for (instr, formula) ← baseset do
13:         result ← result ∪ GENERALIZE(instr, formula)
14:     return baseset

    # learn a formula for instr
15: procedure STRATAONE(instr, baseset, tests)
16:     prog_instr ← INSTANTIATEINSTR(instr)
17:     t = SYNTHESIZE(prog_instr, baseset, tests)
18:     if t is timeout then return ⟨failure⟩
        # the set of equivalence classes of learned programs
19:     eqclasses ← {{t.prog}}
20:     while true do
21:         t = SYNTHESIZE(prog_instr, baseset, tests)
22:         if t is timeout then
23:             break
24:         else
25:             t ← CLASSIFY(t.prog, eqclasses, tests)
26:             if t is failure then return ⟨failure⟩
27:         if |eqclasses| > threshold then
28:             break
29:     bestclass ← CHOOSECLASS(eqclasses)
30:     bestprog ← CHOOSEPROG(bestclass)
31:     formula ← BUILDFORMULA(instr, bestprog)
32:     return ⟨success, formula⟩
```

Algorithm 8 Helper functions

```
    # classify prog into the equivalence classes
 1: procedure Classify(prog, eqclasses, tests)
 2:     eqs ← ∅
 3:     for class ← eqclasses do
 4:         t ← SmtSolver(prog, ChooseProg(class))
 5:         if t is equivalent then
 6:             eqs ← eqs ∪ {class}
 7:         else if t is counterexample then
 8:             tests ← tests{t.counterexample}
 9:             remove all programs p in eqclasses for which
                    Run(p) ≠ Run(prog_instr)
10:             if |eqclasses| = 0 then return ⟨failure⟩
11:     if |eqs| = 0 then
12:         eqclasses ← eqclasses ∪ {prog}
13:     else
14:         eqclasses ← merge all classes in eqs
15:         add prog to merged class
16:     return ⟨success⟩

    # generalize a learned formula for an instruction to a set of formulas
17: procedure Generalize(instr, formula)
18:     result ← ∅
19:     for instr′ ← AllInstructions() do
20:         if instr′ has same mnemonic as instr then
21:             continue
22:         candidate ← None
23:         if operands of instr′ have same size then
24:             candidate ← rename operands in formula
25:         if operands of instr′ have smaller size then
26:             candidate ← rename and sign-extend operands
27:         if operands of instr′ have larger size then
28:             candidate ← rename and select operands
29:         if candidate ≠ None then
30:             test candidate with random constant operands
                      and on random test cases
31:             add to result if all tests pass
32:     return result
```

and Algorithm 8.

We start with a *base set* of instructions, for which we already have formal specifications. This set is meant to be small but should cover all of the unique functionality in the instruction set. For all remaining instructions, we automatically infer a formula for an instruction i by learning a small x86-64 program, consisting only of instructions whose semantics are already known, that behaves identically to i. For instance, consider the instruction variant **decb** (the decrement instruction that operates on an 8 bit register). Since the only operand of **decb** is a register, we can instantiate the instruction (InstantiateInstr in Algorithm 7) to obtain a one instruction program by choosing an input register; e.g., **decb bl**. We call this the *target instruction* t, and attempt to synthesize a loop-free program p_0 that behaves just like t on a number of test cases (more on test case generation in Section 4.2.2) using a stochastic search. For the stochastic search Synthesize we use guided randomized search very similar to the one in the previous chapter. We detail some of the specifics to x86-64 in Section 4.2.8.

For instance, for the decrement instruction we might learn this program:

```
1  xorb al, al # al = 0 and clears cf
2  setae ah    # set ah if cf is 0
3  subb ah, bl # bl = bl - ah
```

The comments indicate the main steps of the program, but note that the instructions not only decrement the **bl** register, but also sets a number of status flags appropriately.

The search for a program has the freedom to overwrite registers that are not written by the target instruction; the resulting program p_0 must only agree with t on the locations that t may write. For example, the program above overwrites **al**, **ah** as well as **cf**, even though the decrement instruction does not affect these locations. Requiring that the synthesized program agree with the target instruction only on the target's write set effectively gives the search temporary scratch locations to use.

Once we have learned a program p, we can convert it to a formula by symbolic execution of p since we have formulas for all the instructions in p. However, the program p may overfit to the test cases, and there might be inputs for which p and t disagree. We run the search again to find a potentially different program p' that also behaves just like t on all test cases. Now, we use an SMT solver to check $p \equiv p'$.

If $p \not\equiv p'$, we get a counterexample, that is, an input for which p and p' behave differently. We can then remove the incorrect program (or possibly both programs) by comparing their behavior on this new input against t. We also add the new test case to our set of test cases and search again. Otherwise, if $p \equiv p'$, we add p' to the set of programs we know. We repeat this process until we can either not find any more programs, or until we have enough programs according to a threshold. We discuss how to deal with solver timeouts and spurious counterexamples in Section 4.2.6.

This process increases our confidence that the learned programs in fact agree with t on all inputs,

and allows us to find tricky corner-case inputs. High confidence is the most we can hope to achieve because, again, our purpose is to infer specifications for which there is in general no ground truth. In Section 3.3.2 we evaluate the correctness of the learned semantics for those instructions where we happen to have hand-written formulas.

Once this processes finishes, we are left with a collection of programs from which we can choose one (details in Section 4.2.7) to use as the semantics of t. At this point, we can add this instruction variant to the base set, and use it in further searches for other instructions. This creates a *stratified* search, where simple instructions are learned first, followed by slightly more complex instructions that use the simpler instructions learned previously, and so on, until finally quite complex instructions are learned.

4.2.1 Base Set

The base set consists of 51 instruction variants that cover the fundamental operations of the x86-64 instruction set:

- **Integer addition** (4 instruction variants).
- **Bitwise operations**, including bitwise or and exclusive or, shifts (both arithmetic and logical) as well as population count (6 instruction variants).
- **Data movement** from one register to another, as well as variants to sign-extend values (7 instruction variants).
- **Conditional move** (1 instruction variant).
- **Conversion operations** between integers and floating-point values for both 32- and 64-bit data types (8 instruction variants).
- **Floating point operations** including addition, subtraction minimum, maximum, division, approximate reciprocal and square root, fused multiply and add/subtract (which is not the same as a multiply followed by an addition due to the higher internal precision of the fused operation), for both single and double precision floating-point values where available, all vectorized (24 instruction variants).
- **Clear registers**. We include `vzeroall`, which clears all vector registers. This is included for technical reasons only: STOKE requires the initial program (conceptually the empty program) to define all output locations, and this instruction allows this easily for the vector registers (1 instruction variant).

Precisely modeling floating-point instructions over bit-vectors in a way that performs well with SMT solvers is challenging [DK14]. For this reason, we model the floating-point instructions in

the base set as uninterpreted functions. This encoding is extremely imprecise, but still provides a useful semantics for many instructions. For example, it is sufficient for inferring that vectorized floating-point addition consists of several floating-point additions over different parts of the vector arguments. The main limitation of using uninterpreted functions is that it complicates equivalence checking. Given two programs that potentially represent the semantics of an unmodeled instruction, the appearance of uninterpreted functions substantially reduces the likelihood that we will be able to show those two programs to be equivalent.

In addition to these instructions, we found that certain data movements are not well supported by any instruction in the x86-64 instruction set. For instance, there is no instruction to set or clear the overflow flag `of` (there is an instruction that can set any of the other four flags), and in fact several instructions are needed just to set this single flag. Similarly, updating, say, the upper 32 bits of a 64-bit general purpose register in a non-destructive manner is surprisingly difficult and again requires several instructions. Many instructions include setting specific flags or parts of registers as part of their functionality, and the fact that the x86-64 instruction set is missing some of these primitives makes implementing such instructions in x86-64 more difficult than necessary. For this reason, we augment the base set with *pseudo instructions* that provide this missing functionality. Of course, like other base set instructions, we provide formulas for the pseudo instructions.

We implement the pseudo instructions as procedures (that could be inlined). Unlike regular instructions, pseudo instructions cannot be parameterized by operands, since assembly-level procedure calls take arguments in predefined locations according to a calling convention. We address this issue by having a template for every pseudo instruction that can be instantiated with several specific register operands. For instance, we might have a template to set a flag that is instantiated separately for all six status flags. We add the following pseudo instruction templates:

- **Split and combine registers**. Move the value of a $2n$-bit register into two n-bit registers (upper and lower half), as well as the inverse operation (2 templates, 108 instantiations).
- **More splitting and combining**. For the 128-bit registers, we also allow them to be split into four 32-bit registers, and vice versa (2 templates, 18 instantiations).
- **Moving a single byte**. Move the value in a 1-byte register to a specific byte in an n-byte register, and vice versa (2 templates, 152 instantiations).
- **Move status flag** to a register and back. Move the bit stored in a status flag to the least significant bit of a register by zero-extending its value, and move the least significant bit of a general purpose register to a specific status flag (2 templates, 22 instantiations).
- **Set and clear status flags** (2 templates, 12 instantiations).
- **Set `sf`, `zf` and `pf` according to result**. The sign, zero and parity flags have relatively consistent meaning across many instructions, and this instruction sets them according to the

value in a given general purpose register (the sign flag is the most significant bit, the zero flag is 1 if and only if the value is zero, and the parity flag indicates if the least significant byte has an even or odd number of set bits in the 8 least significant bits). (1 template, 4 instantiations).

This makes for a total of 11 pseudo instructions with 316 instantiations. All of these have trivial semantics, as they are mostly concerned with data movement.

4.2.2 Test Cases

A test case is a CPU state, i.e., values for all registers, flags and memory. When beginning a synthesis task we initially generate 1024 random test cases. Additionally, we also pick heuristically interesting bit patterns such as 0, -1 (all bits set) and populate different register combinations with these values. Similarly, we pick some special floating-point values, such as NaN (not a number), infinity, or the smallest non-zero value. With 22 heuristically interesting values we generate an additional 5556 test cases for a total of 6580. In Algorithm 7, this step is called GenerateTestCases.

4.2.3 Generalize Learned Programs

So far we have a method for learning a formula for an instruction variant that uses a specific set of registers. We need to generalize these formulas to other registers, and to instructions that take constants (so-called immediates) and memory locations as operands.

4.2.3.1 Generalization to Other Registers

If we have learned a formula for a particular set of register operands, then we can generalize this to other registers by simple renaming. For instance, after learning a formula for **`decb bl`**, we can create a formula for **`decb r8`** by computing the formula for **`decb bl`**. This determines the value for all registers that are written to, including **`bl`**. For all such outputs, we rename all occurrences of **`bl`** to **`r8`**. In our example, we might have learned the formula

$$\begin{aligned} \texttt{bl} &\leftarrow \texttt{bl} - 1_8 \\ \texttt{zf} &\leftarrow (\texttt{bl} - 1_8) = 0_8 \end{aligned}$$

(only showing the zero status flag for brevity; the two updates are done in parallel, so that **`bl`** on the right-hand side refers to the previous value in both cases). Here, $-$ is subtraction on bit-vectors, $=$ equality on bit-vectors, and c_w is the constant bit-vector of width w with value c. This can easily be renamed to obtain the fomula for **`decb r8`**:

$$\begin{aligned} \texttt{r8} &\leftarrow \texttt{r8} - 1_8 \\ \texttt{zf} &\leftarrow (\texttt{r8} - 1_8) = 0_8 \end{aligned}$$

Some instructions read or write implicit locations as well as take explicit operands. For the renaming to be unambiguous, we should learn a formula for an instruction where the implicit and explicit operands aren't referring to the same register. This can be done by ensuring that InstantiateInstr selects the explicit operands to be distinct from the any implicit ones an instruction might have.

This procedure makes two assumptions: (1) that using different registers as operands cannot make the instruction behave differently, and (2) the renaming is unambiguous (after taking care of implicit operands appropriately).

We can validate assumption (1) by running the instruction with different operands and validating that it behaves as we expect. If it does not, we can learn a separate formula for the operands that behave differently. Interestingly, we have found exactly one case where an instruction has different behavior depending on the register arguments, namely **`xchgl eax, eax`**. Normally, **`xchgl`** exchanges two 32-bit registers and clears the upper 32 bits of both corresponding 64-bit registers. However, **`xchgl eax, eax`** leaves everything unchanged[1].

Assumption (2) is actually not true, and the renaming might be ambiguous. For instance, the instruction **`xaddq rax, rcx`** exchanges the two registers and stores the sum in **`rcx`**. That is, the formula for this instruction is (ignoring status flags for simplicity):

$$\begin{aligned} \texttt{rax} &\leftarrow \texttt{rcx} \\ \texttt{rcx} &\leftarrow \texttt{rcx} + \texttt{rax} \end{aligned}$$

where + is the bit-vector addition on 64-bit values. Now, if we want to get the formula for **`xaddq rdx, rdx`** then we have two possible values for updating the register **`rdx`**, as both locations that **`xaddq`** writes to are identical here. Fortunately, this situation only applies to instructions where two or more locations are written that could possibly alias (considering both explicit operands as well as implicit locations), and there are only five mnemonics that write two or more locations: **`cmpxchg`**, **`xchg`**, **`mulxl`**, **`vinsertps`**, and **`xaddw`**. We manually decide the correct value for these cases by consulting the Intel manual as well as testing the instruction on sample inputs. For the first two it does not matter as both possible values are the same, and for the remaining three it is easy to determine the correct value. For instance, the **`xaddq`** instruction writes the sum, and so **`xaddq rdx, rdx`** would store

$$\texttt{rdx} + \texttt{rdx}$$

in **`rdx`**. This generalization is done in BuildFormula in Algorithm 7.

4.2.3.2 Generalizing to Memory Operands

The vast majority of instruction variants that can take a memory operand have a corresponding variant that takes only register operands. We can validate that the two instructions behave identically

[1]To be precise, there are several possible encodings for the instruction **`xchgl eax, eax`**, and only the two

(except for the operand location) on random test cases, and then use the formula we have learned for the register variant.

For several floating-point operations, the memory location used has a different size. For instance, consider a floating-point addition on two 32-bit values where one of the values is stored in memory: `addss xmm0, (rax)`. The corresponding register-only variant operates on two 128-bit registers (because all `xmm` registers where floating-point values are stored are 128 bits). We can still generalize to these instructions, but only considering the lowest 32 bits from the register variant. Again, we validate this step by testing on random inputs.

While in principle this generalization could also be done in BuildFormula, we do this as a post-processing step after we have learned all register-only formulas, namely in Generalize.

4.2.3.3 Generalizing to Immediate Operands

Similar to memory operands, many instructions with immediate operands have a corresponding instruction that takes only registers. Again we can validate the hypothesis that they behave identically (other than where the inputs come from) and use the same formula. Some instructions do not have a directly corresponding instruction, but first require the constant to be extended to a higher bit width. We might hypothesize that the constant needs to be either sign-extended or zero-extended, but otherwise can use a formula already learned. We test this hypothesis on random inputs when we generalize instructions with immediate operands in Generalize. We find that sign-extending the constants is the correct decision for all such instructions (even ones where one might expect a zero-extension, such as bit-wise instructions).

However, other instructions with immediate operands do not have a corresponding register-only instruction. Unfortunately, we cannot easily apply the same trick we used for registers and learn a formula directly for such variants. The problem is we would need to convert the instruction into a program in InstantiateInstr and therefore instantiate all operands. This means that we cannot vary the value of the immediate, and thus can only learn a formula for a particular constant. However, for many cases this turns out to be enough, as we can just learn a separate formula for every possible value of the constant. If the constant only has 8 bits, then brute force enumeration of all 256 possible values is feasible. This approach works well, as many instructions with 8 bit constants actually are a family of instructions and the constant controls which member of the family is desired. For instance, the `pshufd` instruction rearranges the four 32-bit values in a 128-bit register according to control bits in an 8-bit immediate argument. Not only is it possible to learn 256 different formulas for such instructions it is also considerably simpler to only learn a formula for a single control constant rather than one that works for all possible constants.

---

variants that hard-code one of the operands to be `eax` are affected. These essentially encode to `nop`.

4.2.4 Preventing Formula Blowup

Due to the stratified search, where formulas for entire programs are reused in the semantics of newly learned instructions, our formulas have the potential to grow in size and the formulas that are learned later in the process might become very large. We found that a small number of simplifications reduce the size of the synthesized formulas dramatically and cause our learned formulas to be manageable in size.

A toy example illustrates where some of the formula blowup originates from, and how our simplification rules help. Consider the following program that the program synthesis might learn for **addw %cx, %bx**. It uses an add-with-carry operation for the addition, and moves some of the values through several registers before performing the actual operation.

```
1 xorw %ax, %ax    # set cf to 0
2 movw %cx, %dx
3 movw %dx, %cx
4 adcw %cx, %bx    # add with carry
```

This example illustrates two common sources of formula blowup. Firstly, data movements cause superfluous concatenations and bit selects. In the example, lines 2 and 3 move **cx** to **dx** and back, which should not have any effect. However, after line 3, the value for **cx** is now as follows:

$$(\texttt{rdx}[63{:}16] \circ \texttt{rcx}[15{:}0])\,[15{:}0]$$

where $\circ$ represents bit-vector concatenation, and $x[a{:}b]$ selects bits a through b from the bit-vector x. This is because 16 bit registers like **cx** are modeled as selecting the correct bits from the full register (**rcx** here). The second common cause is when constants are used in the programs learned. In the example above, the add-with-carry instruction is used in a context where the carry flag is always 0. Every such constant makes the resulting circuit unnecessarily large.

We preform two types of simplifications:

- Constant propagation.
- Move bit selection over bit-wise operations and across concatenation. For instance, for a 32 bit variable **eax**, we can simplify (**eax** $\circ$ **eax**)[31:0] to **eax** (where $\circ$ is the concatenation operator for bit-vectors). This transformation can sometimes directly simplify the formula, or enable further simplifications in subexpressions.

4.2.5 Finding Precise Formulas

We strongly prefer precise formulas that do not involve uninterpreted functions. To ensure we learn a precise formula whenever possible, we first attempt to synthesize any formula. If the formula is

imprecise (i.e., involves an uninterpreted function), then we additionally perform another search where we restrict the set of available instructions to only ones that have a precise formula. If the search succeeds, we categorize the program as usual and keep learning more precise programs until no more can be found or a threshold is reached. If no program using only this limited set of instructions can be found, then we accept the imprecise formula. For brevity this additional step is not shown in Algorithm 7.

4.2.6 Unknown Solver Answers

An SMT solver (SmtSolver in the pseudo code) does not always answer "equivalent" or "counterexample" when asked whether two formulas are equivalent. It is possible for the solver to time out, or to report that the two programs might not be equivalent, without giving a valid counterexample. The latter can happen with imprecise formulas that make use of uninterpreted functions. In that case, the counterexample provided by the SMT solver might be incorrect: the two programs in question may actually behave identically on that input.

Since we do not have precise formulas for most floating-point operations, we must deal with the fact that we cannot always prove equivalence or inequivalence of programs, and we cannot always obtain useful counterexamples. Instead of learning a set of programs for which we know that all programs are equivalent, we instead learn a set of equivalence classes $c_1, \ldots, c_n$ (called *eqclasses* in StrataOne of Algorithm 7). All programs in a given class are formally equivalent (as proven by the SMT solver). Now, if we learn a new program p, we can try to prove it equivalent to the programs in all of equivalence classes and either add it to an existing class, merge classes if it happens that p is equivalent to programs in currently distinct equivalence classes, or create a new class for just p (done in Classify in Algorithm 8).

Thus, at the end of our search, we have a set of equivalence classes to choose from. If the formulas are precise, then there will typically be only a single class (unless timeouts are encountered, e.g., due to non-linear arithmetic).

4.2.7 Choosing a Program

Once we can no longer find new programs (or we have decided we have found sufficiently many programs), we must pick one program as the representative we use to create a formula. If there is only a single equivalence class, then all programs in it are equivalent and in theory it doesn't matter which program we pick. However, some formulas might be more desirable than others, e.g., if they are simpler or use no non-linear arithmetic. For this reason, we heuristically rank all formulas (that correspond to the synthesized programs) in order of the following criteria.

- Number of uninterpreted functions
- Number of non-linear arithmetic operations

- Number of nodes in the AST of the formula

This means we prefer precise formulas (ones that do not contain uninterpreted functions), formulas that perform well in SMT queries (no non-linear arithmetic that is prone to timeouts) and are simple (small formulas). This step is done in ChooseProg in Algorithm 7.

Since we have no way to further distinguish between equivalence classes, we consider all classes that are above a threshold size (to avoid unreproducible outcomes) and pick the best one according to the ranking heuristic (ChooseClass above).

4.2.8 Guided Randomized Search for x86-64

The guided randomized search used in this chapter is a tool called STOKE [SSA13]. Just like in the previous chapter, this search is inspired by the Metropolis-Hastings algorithm, but simplification from Section 2.1 is not valid since the proposal distribution here is not symmetric, and thus some of the formal guarantees of the Metropolis-Hastings algorithm may not hold.

STOKE starts from an empty program, and uses guided randomized search to try and find a x86-64program that matches the instruction for which a specification should be learned on all test cases. The cost function simply counts the number of incorrect bits on all test cases.

Due to its stochastic nature and the large size of the x86-64 ISA, multiple runs of STOKE usually lead to distinct synthesized programs. This property is useful for synthesizing the multiple programs required by our approach. To complete avoid synthesizing the same implementation several times, we add a term to the correctness cost function that is 1 if the current program has been learned already, and 0 otherwise.

4.3 Evaluation

In this section we evaluate our implementation Strata according to the following research questions.

(RQ4.1) What number of instructions can Strata automatically synthesize a formula for?
(RQ4.2) Are the synthesized formulas correct?
(RQ4.3) How large are the synthesized formulas compared manually written ones?
(RQ4.4) How precise and usable are the synthesized formulas compared to the hand-written ones?
(RQ4.5) Is the simplification step effective?
(RQ4.6) Is the stratification of the synthesis necessary?

We also give some details on the overall synthesis process, such as how long it takes and what fraction of the SMT solver calls returned unsatisfiable, a counterexample or timed out.

Base set	51
Pseudo instruction templates	11
Register-only variants learned	692
Generalized (same-sized operands)	796
Generalized (extending operands)	81
Generalized (shrinking operands)	107
8-bit constant instructions learned	119.42
Learned formulas in total	**1795.42**

Table 4.1: Overall synthesis results. The table lists the number of base instructions, and how many formulas can be automatically learned from these, broken up into different categories.

4.3.1 Synthesis Results

Table 4.1 summarizes the number of formulas that our approach can learn automatically. Starting from 51 base instructions and 11 pseudo instruction templates, both with formulas written by hand, we first learn 692 instruction variants that only use registers as operands. These can then be generalized to other instruction variants that use memory or immediate operands. More precisely, we can generalize to another 796 instruction variants that use operands of the same size as the register-only variants. 81 variants require an operand to be sign-extended and another 107 variants are instructions that use smaller operands (a 32- or 64-bit memory location instead of a 128-bit `xmm` register). Finally, we also learned formulas for instructions with an 8-bit constant that do not have a corresponding register-only instruction. We attempted to learn a separate formula for all 256 different possible values for the constant. In some cases, we learned a formula only for some of the possible constants, which is why we count every formula as 1/256 of an instruction variant. We learn 119.42 instruction variants in this way. In total, Strata learned 1,795.42 formulas automatically, or 61.5% of the instructions in scope.

4.3.2 Limitations

We manually inspected the instructions for which Strata was unable to learn a formula. They can be roughly categorized as follows, though there might be some overlap between categories:

- **Missing base set instructions.** Some instructions perform floating-point operations that do not appear in the base set. The same is true of several vectorized integer instructions. For these instructions, there is no non-vectorized instruction that performs the corresponding primitive operation (e.g., saturated addition). A possible solution would be to add pseudo instructions that perform these operations in a non-vectorized fashion.
- **Program synthesis limits.** Some instructions are learnable in principle, though the length of the shortest program required to do so is beyond the reach of STOKE's program synthesis engine.

- **Cost function.** There are several instructions for which STOKE's cost function does not provide sufficient guidance for search to proceed effectively. For instance, for some instructions it is difficult to produce a program that correctly sets the overflow flag. STOKE's cost function cannot provide much feedback for just this single bit until a mostly correct program is found.

4.3.3 Correctness

STOKE already contains formulas for a subset of the x86-64 instruction set. These are written against the same model of the CPU state used in this chapter, and so we can easily compare the STOKE formulas against the ones learned by Strata by asking an SMT solver if the formulas are equivalent.

We have hand-written formulas from STOKE for 1,431.91 instructions variants (or 79.75%) of the formulas learned by Strata (none of these were instruction variants with a constant). For 1,377.91 instruction variants, we could prove equivalence automatically. We manually inspected the 54 instruction variants where the SMT solver was not able to automatically prove the equivalence between the manually written formula and the one synthesized by Strata. For 4 instructions, the formulas are in fact equivalent if some additional axioms about the uninterpreted functions are added: Because we encode all floating point instructions as uninterpreted functions, the SMT solver does not know that there is a well-defined relationship between some of these. For instance, a fused multiply-add with an additive operand that is zero is equivalent to a regular multiply. Adding such axioms allows us to prove the equivalence for these 4 instructions. Alternatively, if an encoding for floating point operations would be used that the SMT solver can reason about, then no axiom would be necessary.

In the remaining 50 cases, there were semantic differences between the hand-written and synthesized formulas. We inspected each case manually by consulting the Intel manual as well as running example inputs on real hardware to determine which is correct. In all cases, the formulas learned by Strata were correct and the manually written ones were not:

- The AVX extension adds many variants of existing instructions with three operands; two source operands and one destination. In 32 cases, the manually written formulas incorrectly left the upper 64 bits of the destination unchanged instead of copying the bits from the first source operand.

- For 10 vectorized conversion operations (from double to single precision floating point values or from doubles to 32-bit integers), the upper 64 bits of the result are supposed to be 0, but the hand-written formulas mistakenly left these unchanged.

- There are 8 variants of a combined addition/subtraction operation (e.g., `addsubps`). The hand-written formulas erroneously swap the position of the addition and subtraction operations.

In summary, we found 50 instruction variants where the hand-written formulas contained errors, and 4 cases where the formulas required some relatively simple axioms about the uninterpreted functions used to be proven equivalent. All other automatically learned formulas are provably equivalent to the hand-written ones.

Errors in Intel documentation The Intel manual [Int15] only contains informal pseudo-code and an English description of the semantics, so there is no automatic way of comparing the learned formulas with the manual. However, over the course of this project we found several inconsistencies when manually comparing the two. This is unsurprising given the over 3,800 pages of intricate technical content, the vast majority of which is not machine-checked. Examples of these errors include:

- The upper 128 bits of the output register of **`pshufb xmm0, xmm1`** are kept unmodified on Intel hardware, but the pseudo-code in the manual incorrectly says they are cleared.
- The *Op/En* column for the **`vextracti128`** instruction should be MRI, but it was RMI. This causes a wrong operand ordering when assembling this instruction.
- One possible encoding for **`xchgl eax, eax`** is `0x90` according to the manual, and its semantics are to clear the upper 32 bits of **`rax`**. However, `0x90` is also the encoding of **`nop`**, which does nothing according to the manual (and hardware).
- The textual description of `roundpd` claims to produce a single-precision floating point value, when it produces a double-precision floating point value.

We have reported all of these items and they were confirmed by Intel as errors in the manual [Cha16].

4.3.4 Formula Size

Large formulas can cause problems for SMT solvers and perform less well, especially if formulas are further combined (e.g., to prove equivalence between two x86-64 programs). For this reason, we answer (RQ4.3)by comparing the number of nodes in the AST of the formula representation in the automatically learned formulas and the hand-written formulas. For every instruction variant where we have both a learned and hand-written formula, we compute

$$\frac{\text{number of nodes in learned formula}}{\text{number of nodes in hand-written formula}}$$

We plot a histogram of this distribution in logarithmic space (such that increases and decreases in size are shown symmetrically) in Figure 4.1. The center of the figure consists of formulas that have the same size, with instructions where Strata's formulas are smaller to the left and instructions where the hand-written formulas are smaller to the right. The median of this distribution is one (i.e., formulas are equally large). The smallest formula has size 0 (for the **`nop`** instruction), and the largest formula has 211 and 196 nodes for Strata and the hand-written formulas, respectively.

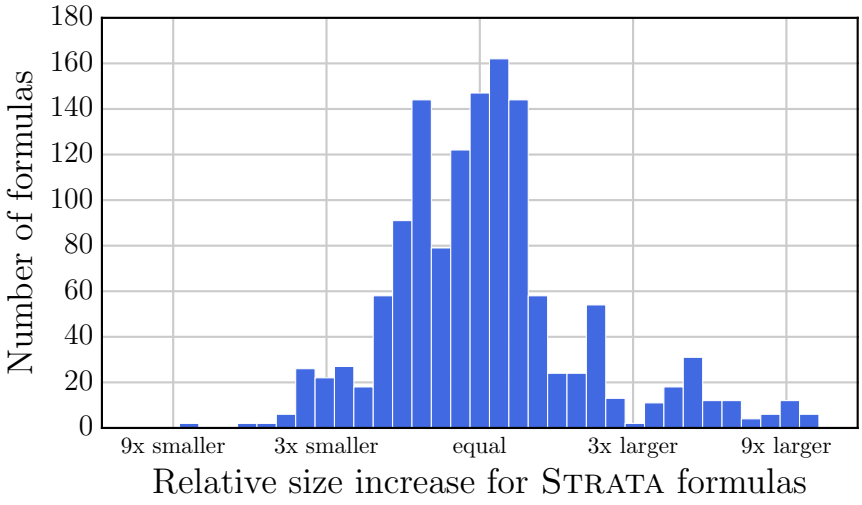

Figure 4.1: Distribution of the relative increase in size of the formulas learned by STRATA compared to the hand-written formulas. Negative numbers indicate that the learned formulas were smaller.

4.3.5 Formula Precision

Two further important aspects of the quality of formulas is the number of uninterpreted function applications (a measure of preciseness) and the number of non-linear arithmetic operations (which tend to perform poorly in queries), as stated by (RQ4.4). We found that the automatically synthesized formulas have the same number of non-linear operations, except for 9 instruction variants, where they contain fewer such operations (the hand-written formulas for some rotate instructions contain a modulo operation, whereas the automatically learned ones do not). In all of these cases the non-linear operation is a modulo by a constant, and is likely not a big problem for SMT solvers (unlike non-linear operations by variables).

The number of uninterpreted functions (UIF) is the same for almost all instruction variants, with the exception of 4 formulas, where the automatically learned formulas contain redundant UIFs: Instead of using a single UIF, the learned formula uses 2 UIFs that combined perform the same operation as the single UIF. In all cases a simplification step that knows about floating point operations should be able to remove these redundant UIFs.

In summary, the automatically learned formulas are comparable in size, and with the exception of 4 instruction variants are equally precise and contain at most as many non-linear arithmetic operations compared to the hand-written formulas.

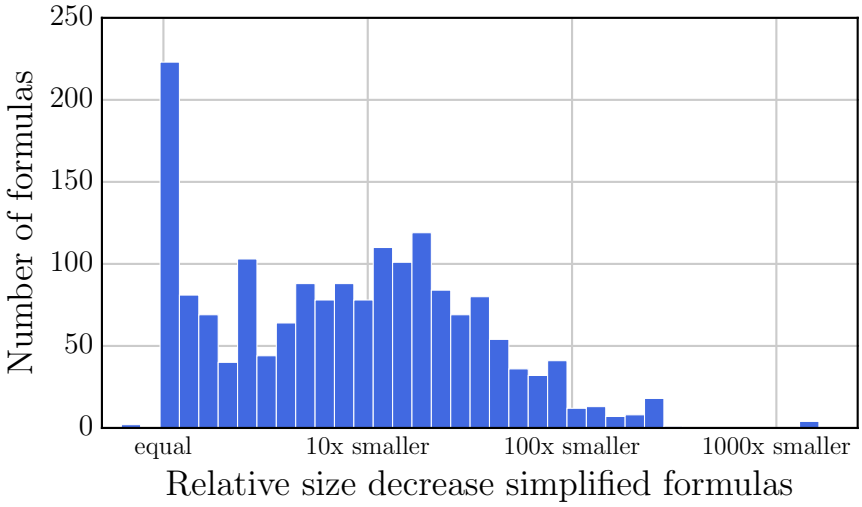

Figure 4.2: Distribution of the relative decrease in size of the formulas learned by STRATA due to the simplification step.

4.3.6 Simplification

To address (RQ4.5), we measure the size of the learned formulas before and after simplification and compute the relative decrease in size (as before when comparing with handwritten formulas). At the median of this distribution, simplified formulas are 8.45 times smaller, and in the extreme they are smaller by up to a factor of 1,583. A small number of formulas are larger after simplification (due to using a distributive law), but at most by a factor of 1.33. The full distribution is shown in Figure 4.2.

4.3.7 Stratification

To answer (RQ4.6), we can investigate if instructions could have been learned sooner. To this end, we define the *stratum* of a formula that corresponds to an instruction variant i inductively as follows. Let $M(i)$ represent the set of instructions that the synthesized program for i uses, and let B be the set of instructions in the base set. Then,

$$\text{stratum}(i) = \begin{cases} 0 & \text{if } i \in B \\ 1 + \max\limits_{i' \in M(i)} \text{stratum}(i') & \text{otherwise} \end{cases}$$

Intuitively, this captures the earliest point at which a particular formula could have been learned with the given program. We show the distribution of strata in Figure 4.3, which shows that stratified synthesis is in fact necessary for our approach to learning the semantics of x86-64: only 13.73% of

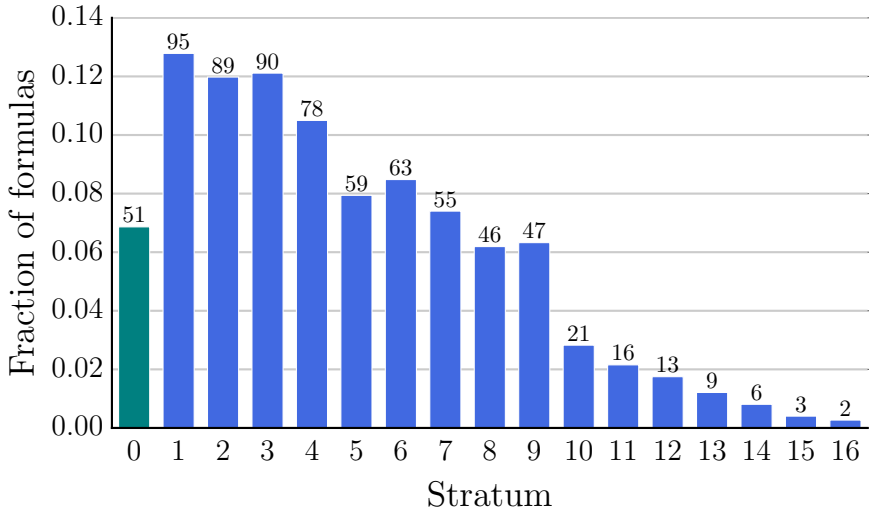

Figure 4.3: Distribution of the strata for all register-only instruction variants. Formulas at stratum 0 are from the base set, and all others are learned formulas.

instructions are learned at stratum 1, the 50th percentile of all learned instructions is reached only at stratum 4, and the 90th percentile is not reached until stratum 10.

In Strata, formulas are added to the base set as soon as a specification has been learned (by learning multiple programs). So when an instruction i at stratum n is added to the base set and another instruction i' makes use of i during the search for its specification, we say i' was learned at stratum (at least) $n + 1$. However, there might be another (equivalent) specification where the instruction i was not used, and thus the stratum for i' then might be different (and lower). Therefore, to definitively answer (RQ4.6), we run Strata where new instructions are not added to the base set (for the same amount of time, on the same machine as the main experiment). This effectively turns off stratification and determines the maximum number of instructions this approach can learn at stratum 1. We find that for the register-only variants, we can learn only 426 formulas without stratification, compared to the total 692 formulas learned with stratification (an increase of 62.4%). In the experiment with stratification, only 95 formulas were learned at stratum 1, showing that indeed, some formulas at higher strata could have been learned at lower strata if we had given more time to the program synthesis at those lower strata. However, overall stratification was essential to learning.

4.3.8 Experimental Details

We performed all experiments on an Intel Xeon E5-2697 machine with 28 physical cores, running at 2.6 GHz. For engineering reasons, we split the problem into learning register-only variants and

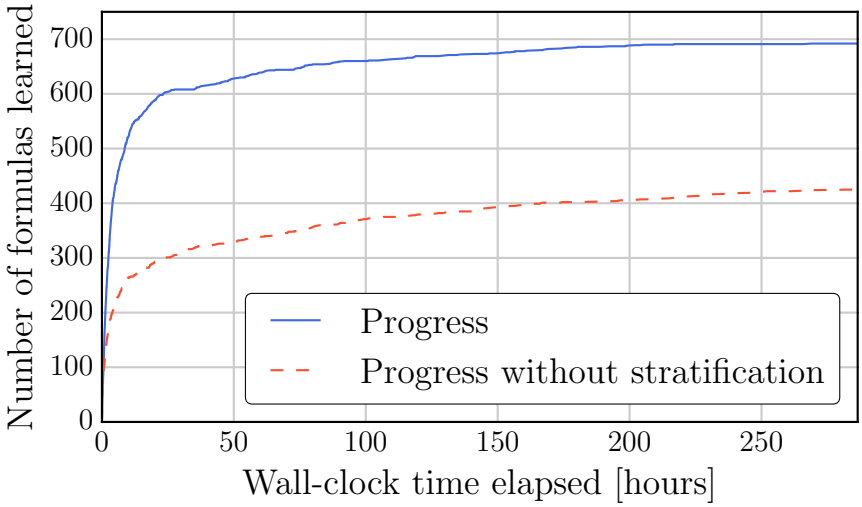

Figure 4.4: Progress over time as measured by the number of (register-only) formulas learned.

learning formulas for instructions with an 8-bit constant. Finally, we generalized these formulas to other similar instruction variants with memory or immediate operands.

Our algorithm is embarrassingly parallel and we run it on all available cores. We let the register-only experiment run for 268.86 hours, and spent 159.12 hours for the 8-bit constant experiment. Total runtime therefore is 427.98 hours, or 11983.37 CPU core hours. Figure 4.4 shows the progress during the learning of register-only instructions, both for Strata as presented and Strata with stratification turned off. Strata is able to quickly learn formulas for the simple instructions, with progress tapering off as more difficult instructions are attempted.

During the process of learning formulas, we invoked an SMT solver to decide what equivalence class a program belongs to (procedure Classify in Algorithm 8). Out of the 7,075 such decisions Strata had to make when learning register-only variants, we found an equivalent program in 6,669 cases (94.26%) and created a new equivalence class in 356 cases (5.03%) as the solver could not prove equivalence with any existing class. In only 3 out of these 356 cases a solver timeout was the cause, and in all other cases the solver did not give a valid counterexample.

We also found a (valid) counterexample in 50 cases. This highlights the importance of learning many programs, as the first one we learn can actually be wrong. It also shows that for the most part the testcases work well.

4.3.9 Implementation

We make Strata, our implementation of stratified synthesis available online, together with the results of the experiments described in this chapter including all learned formulas.

https://stefanheule.com/strata/

4.4 Related Work

There has been a proliferation of program synthesis techniques in recent years. These include techniques based on SAT solvers [SRBE05], SMT solvers [JGST10, GJTV11], exhaustive enumeration [BA06, SR15, FCD15, OZ15], and stochastic search [SSA13, SSA14, NORV15]. In this work, we demonstrate that stratification significantly enhances the performance of STOKE, a stochastic synthesizer of x86-64 code. We believe that other approaches to program synthesis can also benefit from the modularity introduced by stratification, which we leave as future work. Our focus in the current work is to obtain a formal semantics for x86-64 instructions.

The work closest to ours is by Godefroid and Taly [GT12]. Using 6 manually provided templates and SMT-based synthesis, they are able to synthesize formulas for 534 32-bit x86 arithmetic (no floating-point) instructions. For tractability, the templates are required to have "smart inputs", which is a set of inputs such that if two instantiations of the template agree on the smart inputs then the instantiations are guaranteed to be equivalent for all inputs. They show that in the absence of smart inputs, the SMT based synthesizer runs out of memory while synthesizing formulas for 32-bit shifts [GT12]. In contrast, we do not require templates, smart or otherwise. We bootstrap a small base set of formulas to obtain formulas for a larger set of instructions. While the base set of formulas that we use were written manually, many of these could be synthesized by extending the approach of [GT12] to x86-64.

The HOIST system automatically synthesizes abstract semantics (for abstract domains such as intervals, bit-wise domain, etc.) by exhaustively sampling 8-bit instructions of embedded processors [RR04]. In subsequent work, templates were used to scale this process to 32-bit instructions [RD06]. In contrast, we synthesize bit-precise concrete semantics for x86-64 instructions that operate on registers which can have up to 256 bits. These concrete semantics can be used to obtain abstract semantics (for any abstract domain) using [RSY04].

Several research groups have implemented, with considerable effort, their own formal specification for the x86 instruction set. Our work automates this task and generates a more complete specification than the existing ones. For instance, CompCert includes a partial formal specification for 32-bit x86. The paper [SR15] uses formulas for 179 32-bit x86 opcodes described in [LR13]. The STOKE tool itself contains manually written formulas for a subset of x86-64. Other platforms for x86 include BAP [BJAS11], BitBlaze [SBY+08], Codesurfer/x86 [BGRT05], McVeto [TLL+10], Jakstab [KV08], RockSalt [MTT+12], among many others. We note that Intel does not appear to have a formal model that fully defines CPU behavior, not even internally [ATS+15, Section 4.1].

The idea of comparing two implementations by translating them to SAT or SMT solvers has been used in many contexts, e.g., deviation detection to find bugs [RE11] or to determine the soundness

of optimizations [BA06]. In our work, we use it to find corner-case inputs that are not part of our testcases.

Weitz et al. [WLH+17] presents SpaceSearch, a library to build solve-aided tools within the proof assistant Coq and the ability to extract the tool to a solver-aided language like Rosette [TB13], which can make use of efficient SMT solvers such as Z3 [WHdM13]. As part of the evaluation of SpaceSearch, the authors compare the x86 semantics of Rocksalt [MTT+12] (a formal checker for the safety of Native Client [YSD+09]) with the semantic we learned. Initially 72.7% of all instruction variants tested had at least 1 bit in the output that was inconsistent between the two semantics. After investigation, the authors traced the discrepancies to 7 root causes in RockSalt, and 1 root cause in Strata. Strata learned the correct formula, but when it was printed to be used by SpaceSearch, the pretty printer contained a bug when simplifying the formulas as part of the printing. The bug is only triggered during pretty printing, not during regular use of the learned formulas. This further validates that writing specifications is difficult by hand, and that Strata can effectively learn the specifications.

Since we published this work as [HSSA16], Hasabnis et al. [HS16] have proposed another way of automatically learning a semantics for an instruction set: This work proposes symbolically analyzing code generators as they are found in modern compilers to learn the semantics of instructions. This has the advantage that after an upfront investment in building the symbolic analyzer it is easy to learn the semantics of another instruction set, as long as the code generator supports that instruction set. This approach relies on the correctness of the code generator, and is less precise in some respects (for instance in learning the semantics of status flags). However, compared to our work, new architectures can easily be supported and can handle most instructions that the code generator supports.

Chapter 5

Improving Stochastic Search for Program Optimization

Auch ein blindes Huhn findet 'mal ein Korn.
– German proverb
(Even a blind chicken sometimes finds a grain.)

5.1 Overview

Producing fast executable code remains an important problem in many domains. Significant engineering efforts have gone into modern optimizing compilers, yet for the most performance critical applications software developers still often choose to hand-write assembly functions. A major reason why optimizing compilers cannot compete with hand-tuned assembly is the complexity of modern CPU architectures such as x86-64. It is notoriously difficult to estimate the performance of a piece of code statically, making it difficult for compilers to choose one sequence of instructions over another. There are many subtle effects, historical artifacts and surprising performance characteristics in modern processors. The situation is further complicated by the fact that the details of CPU architecture internals are closely guarded secrets; for example, while it is known that many processors split instructions into micro-ops, the details are largely unknown. Overall, developing compilers that produce the best possible code is still a challenge.

To illustrate this point, consider the program in Figure 5.1, which calculates the number of bits set in a buffer of `n` elements. The implementation makes use of an intrinsic to use the **`popcntq`** instruction available on modern CPUs, which can calculate the number of set bits directly on an 8-byte value (the so-called *population count*). One possible implementation on x86-64 is shown in

```
1 #include <x86intrin.h>
2 uint32_t run(uint64_t* buf, int n) {
3   uint64_t res = 0;
4   for (int i = 0; i < n; i += 1) {
5     res += _mm_popcnt_u64(buf[i]);
6   }
7   return res;
8 }
```

Figure 5.1: C++ program to calculate the number of bits set in a buffer.

Figure 5.2. The instruction on line 7, **xorq %rdx, %rdx**, is functionally irrelevant: it sets the register **rdx** to 0, right before the value of **rdx** is overwritten by the next instruction when computing the population count. However, if we benchmark the program with and without the useless instruction on an Intel Haswell CPU, the program without the seemingly superfluous instruction runs at 9.55 GB/s, while the program with the instruction runs at 19.09 GB/s, an increase of almost 2x in performance for adding an instruction with no obvious benefit. To the best of our knowledge, this behavior arises because the CPU's dynamic instruction scheduling detects a false dependency of **popcntq** on its destination register. When executing the instruction **popcntq %rax, %rcx** (which computes the population count of **rax** and stores it in **rcx**), the CPU (unnecessarily) waits for the value in **rcx** to be available, even if it is only written, which creates a loop carried dependency. Adding the **xorq** instruction breaks this dependency, as the CPU realizes it does not need to wait for the previous iteration to finish. Recent compilers have hard-coded knowledge of this particular oddity (and thus it does not show up as a performance opportunity in any of our experiments), but there are countless other performance anomalies, often less severe but potentially still exploitable. We believe it is extremely unlikely that compilers will acquire hand-coded knowledge of all such interesting performance artifacts.

A radically different approach is taken by stochastic superoptimization [SSA13] that uses the Metropolis-Hastings algorithm described in Chapter 2. The cost function here is going to measure both correctness as well as performance, eventually trying to find a program that is correct, and as fast as possible.

Guided randomized search promises to avoid the issue of complex trade-offs between different instruction sequences by using the cost function to estimate the performance and letting the search find a good trade-off. For optimization this strategy relies on having a good estimator of program performance. Previous work on stochastic superoptimization [SSA13] uses a *latency cost function* that sums the latencies of a program's instructions as the performance estimate. The latencies of all instructions are determined beforehand and stored in a lookup table, making the latency cost function very cheap to compute. However, such a simple cost function cannot capture the complex performance characteristics discussed above. For instance, not only will the effect of removing the

```
1  # arguments: rdi = buf, rsi = n
2  .count_set_bits:
3    leal -0x1(%rsi), %eax          # rax = n-1
4    leaq 0x8(,%rax,8), %rcx        # rcx = n*8
5    xorl %eax, %eax                # rax = 0 (res)
6    addq %rdi, %rcx                # rcx = buf + n*8 (end of buffer)
7  .loop_body:
8    xorq %rdx, %rdx          # (*) # rdx = 0
9    popcntq (%rdi), %rdx           # rdx = popcnt
10   addq %rdx, %rax                # rax += rdx
11   addq $0x8, %rdi                # rdi += 8
12   cmpq %rcx, %rdi                # end of loop?
13   jne .loop_body
14   retq
```

Figure 5.2: Possible x86-64 implementation of Figure 5.1. Removing the instruction marked with (*) results in a slowdown of almost 2x on some architectures, even though the instruction does not contribute to the calculation.

xorq instruction from the program in Figure 5.2 be invisible to this estimate, it will predict the version with the additional **xorq** instruction is slightly slower (instead of 2x faster).

In this chapter we propose a different cost function of just running the proposed program on a benchmark input and measuring its execution time. The straightforward nature of the idea disguises a number of non-trivial challenges to realizing it in practice, such as handling programs that crash or instrumenting the code to measure its execution time in the presence of position dependent code. This new cost function, which we call *realtime cost*, is easily able to correctly identify the faster of the two versions of the program in Figure 5.2 and promises to search the space of all programs much more effectively by more precisely characterizing the fast programs.

We evaluate both the latency and realtime cost functions on a set of 10 benchmark programs that use diverse subsets of the available x64-86 instruction set. We find that guided randomized search behaves radically differently on different programs, and requires a careful choice of search parameters. Furthermore we show that the latency estimate cost poorly predicts actual performance, and that this poor estimate leads to spending large fractions of the search on poorly performing parts of the search space. We show that for our benchmarks the realtime cost function is able to find a given improvement over the baseline more quickly than the latency cost. Furthermore, for several benchmarks the realtime cost function is able to find programs that are significantly faster than the latency cost function finds. Perhaps surprisingly, we also show that for some benchmarks, the fastest programs found by the two cost functions are roughly equally fast (see discussion in Section 5.3.5).

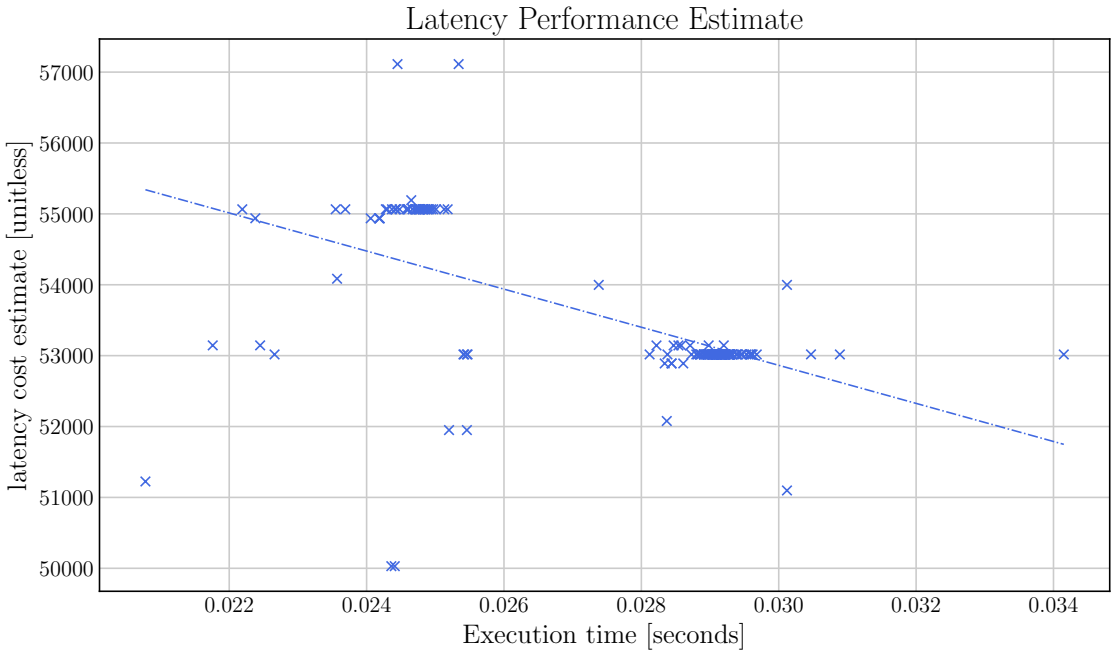

Figure 5.3: Scatter plot of 160 variants of a program, showing benchmarked execution time against latency cost estimate, as well as a linear regression of these data points.

5.2 Approach

As discussed previously, in STOKE, the performance estimate is defined as the sum of latencies of all instructions in a program:

$$\text{perf}_S(p) = \sum_{i \in \text{inst}(p)} \text{latency}(i)$$

We have found this estimate to be rather disappointing in accuracy. It does not take any dependencies between instructions into account, nor are any of the performance intricacies (such as the one highlighted in Section 5.1) considered. To underline this problem, we benchmarked 160 variants of a program[1] and plot the benchmarked execution time against the latency cost estimate. For a perfect predictor, one would expect perfect linear correlation, however as Figure 5.3 shows, the latency cost is a rather poor estimate. In fact, the linear regression has negative slope, and the Pearson Correlation Coefficient (a measure of correlation) is -0.58.

We propose a *realtime* estimate, where we run the program p on bare metal. In the remainder of this section we outline how this idea can be implemented, and point out several of the technical challenges of doing so.

One might wonder why STOKE doesn't already use actual performance as its metric, given that it

[1]The exact benchmark used is less relevant, but for completeness, it is the *karim* program, see Section 5.3.

already needs to run the program p to determine its correctness. The problem is that the program p is run in a sandbox to avoid corrupting the STOKE process if the program p crashes, for instance by reading from inaccessible memory or by performing another illegal action such as a division by zero. STOKE uses binary translation to instrument potentially problematic instructions and checks that all inputs to the instruction are valid. This sandboxing is crucial for the correct execution of STOKE, but also prevents measuring the runtime of p by simply measuring the time it takes to execute p in the sandbox. Due to the instrumentation there is a significant difference in the instructions executed on the CPU when running the sandboxed code and just running p directly. Even if one were to rewrite STOKE to use another sandboxing approach (e.g., Intel's virtualization technology), there are likely important differences in the performance behavior of p and the virtualized version of p. Furthermore, care must be taken to ensure that running p will find the testcase memory in the correct location.

5.2.1 Implementation

We now describe how we get accurate performance measurements.

Some x86-64 instructions have behavior that depends on the instruction address, in particular instructions that use so-called instruction pointer-relative addressing, which allows accessing memory locations relative to the current instruction's address. With the binary rewriting approach taken by STOKE, relative addressing can easily be supported by appropriately rewriting these addresses, but if we want to natively execute p we must put the code for p at the address where p is meant to be. This can be difficult to do within STOKE's process directly, as that address might already be used. Instead, we create a minimal background process whose function is to benchmark proposed programs. We compile the code for this process with `-pie`, such that its main routine is allocated at a random address and thus unlikely to use an address needed by p. At initialization time this process allocates the testcase memory (which includes both stack and heap segments), as well as the executable buffer where the code of p will go. Additionally, it allocates two executable buffers which contain a small amount of setup and teardown code that measure the execution time, as well as prepare the process to execute p. Finally, the process waits on a pipe set up with the main STOKE process to receive a proposal p.

When a proposed program p is received (as assembled bytes), it is written to the executable buffer, and then the setup code is invoked. This code is hand-written assembly, which backs up all callee-saved registers on the stack of the background process. The stack pointer **`rsp`** is also saved, but to a fixed known location (we will later change the stack to p's stack, thus we cannot store **`rsp`** on the stack of the background process). Next, the memory from the testcases is initialized, setting up the stack and heap for p. However, the address `(%rsp)` is overwritten with the address of the cleanup-code, so that when p returns, we return to the cleanup code. This change is typically safe to

[2] We specifically check for this property in the STOKE sandbox, and abort if p would modify the location.

do, as that address just contains the return address to whatever code called p, and generally that address is not inspected or used by p.[2] Then, all registers are initialized according to the testcase that is about to be run. This overwrites the stack pointer, effectively switching to the stack of p. Next, the current time-stamp counter of the CPU is saved as the start time (using the `rdtscp` instruction). Finally, using an absolute jump the process starts executing the code of p.

Once p finishes with a **`retq`** instruction, control is handed to the cleanup code, which first measures the time-stamp counter again to obtain the number of cycles p consumed. We then restore the state of the background process by first restoring the **`rsp`** stack pointer and then popping all callee-saved registers. This whole process is run a configurable number of times, measuring the execution time of p several times. Finally, we compute the lowest recorded value and return this value as the cost to the main STOKE process.

The full pseudo-code for the background process is shown as Algorithm 9.

Algorithm 9 realtime cost function

```
1:  procedure REALTIME(p)
2:    Run p in sandboxed fashion
3:    if p does not return normally then
4:      return very high cost # faulting programs are not evaluated for speed
5:    if p read or wrote (%rsp) then
6:      return very high cost # currently a limitation
7:    a ← assemble(p)
8:    send a to background process
9:    if background process crashed then
10:     restart process # safety-net
11:     return very high cost
12:   wait and return for measured execution time t
13: procedure BackgroundProcess(T, addr)
14:   allocate memory required by testcase T
15:   allocate executable buffer for p at given address addr
16:   allocate executable buffer for setup code
17:   allocate executable buffer for cleanup code
18:   while true do
19:     wait to receive assembled program p
20:     write p to pre-allocated buffer
21:     times ← ∅
22:     for i from 1 to reps do
23:       initialize testcase memory
24:       jump to setup code
25:       times ← times ∪ {recorded time}
26:     send min(times) to STOKE
```

5.2.2 Pitfalls

It is all too easy to get randomized search for program optimization wrong in subtle ways. In this section we highlight some of the pitfalls we encountered.

5.2.2.1 Choosing Search Parameters Correctly

As explained in Chapter 2, the search will accept a proposed program with a certain probability even if it has a higher cost c' than the current program's cost c. This probability is

$$\min\left(1, \exp\left(-\beta \cdot (c' - c)\right)\right)$$

with a user-defined parameter β, which controls how likely it is that a proposal is accepted; the larger β, the less likely the search is to accept bad proposals. Unfortunately this parameter is difficult to understand intuitively, and it is on a logarithmic scale. In addition, when trying to compare different cost functions, or when trying to use the same rate of acceptance for worse proposals across different benchmarks, one must be very careful. The probability is proportional not only to β, but also the difference $c' - c$. This difference might be very different depending on the cost function as well as the particular program to be optimized.

To simplify the choice of this parameter, we replace β with a new parameter γ, which is defined as the probability with which a proposed program that is x higher in cost should be accepted, where x is 10% of the cost of the initial program. This is more intuitive, and we can easily calculate β from γ as follows:

$$\beta = -\frac{\log(\gamma)}{0.1 \cdot c_t}$$

where c_t is the cost of the initial program. Not only is this parameter easier to understand, it is also independent of the magnitude of the cost function and allows us to more easily compare costs of different magnitudes. In Section 5.3 we evaluate the effect of choosing different values for γ on the search.

5.2.2.2 Performance Governor

Modern operating systems allow for different so-called CPU *governors* [gov] that control the power schema of the CPU. This allows for trading off between power usage and computational resources. For instance, on Linux three of the available governors are `performance`, `powersave` and `ondemand`. The first two set the CPU frequency to the highest and lowest available frequency, respectively while `ondemand` sets the frequency based on the current system load, which is estimated by a scheduler.

When measuring the wall-clock time a program takes to execute, it is important that the CPU operates at a constant frequency, which is not guaranteed by the `ondemand` governor. It is thus crucial to use the `performance` governor when taking measurements.

5.2.2.3 Machine Load

Another factor that affects the runtime of programs is whether the load of a machine remains consistent across the whole experiment. We measure thousands of programs, so we decided to run measurements in parallel, ensuring we run with a fixed number of threads using the parallelization tool GNU parallel [Tan11]. Running several measurements in parallel introduces some additional noise, but can also be more realistic if the benchmarked code will be run on a non-idle machine in production. We discovered when comparing two versions of the realtime cost function that nondeterministically one and or the other would find a small number of programs that were significantly faster than all the rest.

It turns out that the machine was under constant load throughout most of the performance measurement, but near the end when there were not enough programs to be run to keep all the threads busy, the remaining programs would begin to run faster. We confirmed that the programs were not inherently faster, it was just that the running time was proportional to machine load. (We believe programs ran faster on a lightly loaded machine because of less memory contention.) Once discovered, the fix was easy: We keep running benchmarks (without recording the results) at the end, until all (real) jobs are complete.

5.2.2.4 Removal of Superfluous Instructions

Random program transformations often introduce extra instructions that have no effect on the computation. Especially if performance is not a concern (e.g., when doing program synthesis rather than optimization), these additional instructions can dominate programs. For this reason, STOKE by default automatically cleans up the resulting programs by removing any instruction that has no side-effects (such as raising a signal) and whose output is not used by the program, as well as unreachable code and no-ops. During the search superfluous instructions are kept, but removed just before saving the result on disk at the end of the search. This is useful to make synthesized programs more readable, but might affect performance gains found during the search. For instance, the program from the introduction gained almost 2x in speed by adding an instruction whose results are not used. It is important to keep the program as is, including any seemingly superfluous instructions, as they might just be what makes the program fast.

5.3 Evaluation

In this section we evaluate STOKE's original latency-based cost function and the real time cost function. As a baseline, we also consider a cost function that measures only correctness. Because there is no guidance from the cost function about performance, this cost function implements a pure random walk through the space of correct programs. We want to answer the following research questions:

(RQ5.1) What is the proper choice for the search parameter γ that determines the acceptance probability of higher-cost programs?

(RQ5.2) What part of the search space is explored by the latency and realtime cost function?

(RQ5.3) Which cost function is able to find faster programs?

(RQ5.4) Which cost function is able to find programs with a given speedup (e.g., 10%) more quickly?

(RQ5.5) How good of a performance estimate are the latency and realtime cost functions?

(RQ5.6) Do the latency and realtime cost functions find similar optimizations?

5.3.1 Benchmark Programs

To evaluate how well STOKE optimizes programs, both using the latency and realtime cost functions, we chose 10 submissions to the nibble sort competition [nib]. Nibble sort takes a 64-bit number, which is interpreted as a vector of sixteen (unsigned) 4-bit numbers (so-called *nibbles*) to be sorted. For instance, the input `0x142b2cb134b50824` should be sorted as `0xcbbb854443222110`. The full task is to sort a buffer of n such values (independently). For our experiments, we chose $n = 128$. This is an attractive benchmark for our purposes because the programs are of interesting length (roughly 50-300 lines of assembly) and diverse in both algorithmic strategy and the specific instructions used. Some implementations use SIMD instructions, making heavy use of vectorization, others use look-up tables, clever bit-level tricks, simpler insertion sorts, or a mix of techniques. This provides for a rich set of benchmark programs that test different aspects of the x86-x64 instruction set.

The original contest had 52 submissions, from which we excluded 2 programs for not correctly implementing nibble sort, 37 submissions due to limitations of STOKE, and 3 programs because they read the top of the stack (which is incompatible with the realtime cost function, see Section 5.2.1). This leaves 10 programs, which we describe in Table 5.1.

5.3.2 Experimental Setup

We use a test harness that runs the different nibble sort implementations on a number of pathological cases (collection of specific fixed nibbles) as well as some random tests. This harness is a straightforward adaptation of the harness from the original contest.[3] STOKE can instrument this harness to collect testcases for every benchmark program. A testcase contains the value of all CPU registers (general purpose registers (**`rax`**, **`rcx`**, etc.), vector registers (**`ymm1`**, **`ymm2`**, etc.) and six status flags (**`af`**, **`pf`**, **`sf`**, **`of`**, **`cf`**, **`zf`**)), all memory locations read by the benchmark (this includes the stack and heap, as well as the buffer of numbers to be sorted). In addition to the testcases gathered from

[3] https://github.com/regehr/nibble-sort/blob/master/test_nibble.c

Benchmark	Implementation strategy	x86-64 LOC	Runtime	γ_{latency}	γ_{realtime}
alexander3	vectorized loop-free implementation using bit-wise operations	143	0.02630s	1%	0.1%
anon	loop-free implementation using bit-wise operations	287	0.10968s	0.01%	0.01%
beekman1	vectorized loop-free implementation using vector shuffles	96	0.02704s	10%	0.01%
bosmans2	insertion sort, inner loop implemented using bit-wise operations	56	0.14639s	1%	0.001%
hans	store nibble counts in `uint64_t` value using a first loop, then build sorted result in a second loop	325	0.10110s	10%	0.001%
karim	nested loop, using bit-masks to count number of occurrences	48	0.12507s	0.1%	0.001%
mentre	store nibble counts in `char` array using a first loop, then build sorted result in a second loop	86	0.09452s	10%	0.01%
payer	store nibble counts in `uint64_t` value using bit-wise operations, then build sorted result in a second loop	149	0.27607s	1%	1%
rogers	store nibble counts in `char` array using a first loop, then build sorted result in a doubly-nested loop	93	0.11374s	0.1%	0.0001%
vetter2	store nibble counts in `char` array using a first loop, then build sorted result in a second loop	244	0.09477s	0.1%	0.1%

Table 5.1: Overview of all benchmark programs used in the evaluation. The runtime indicates baseline performance, and lower is better. γ_{latency} and γ_{realtime} are the best values of γ for the two cost functions, respectively.

the harness, we augment the set with testcases that are identical to the existing ones in all registers that are live on entering the benchmark, but contain random values for all other registers. For the benchmarks, the only live registers are **rdi** (the pointer to the buffer to be sorted), as well as **rsp** (the stack pointer). These tests prevent programs found during the search for being rewarded for relying on accidentally useful values found in dead registers. This yields a total of 501 testcases per benchmark.

We perform program optimization searches with STOKE. Each search ran for 1 million iterations, starting from a benchmark program compiled with `gcc -O3`. We first performed 40 searches for each benchmark and for the latency and realtime cost functions with 8 different, logarithmically spaced values for γ:

$$0\%, 0.0001\%, 0.001\%, 0.01\%, 0.1\%, 1\%, 10\%, 100\%$$

We selected for each benchmark and cost function individually the value of γ that found the most programs faster than the baseline (the best values for γ are shown in Table 5.1). We then ran 100 searches for every benchmark and cost function combination using the chosen values of γ.

All experiments were performed on Intel Xeon E5-2697 machines with 28 physical cores clocked at 2.6 GHz, running Ubuntu 14.04.

For all our experiments, we only consider programs that pass all testcases. Given a proposed program, we evaluate its performance by replacing the implementation in the harness with the assembled rewrite (drop-in replacement) and measuring its performance by running 20000 iterations of sorting a buffer. We repeat this measurement 5 times and average the running time. We use a 1-second timeout for the measurement, which is several times longer than the baseline performance of all benchmarks.

5.3.3 Search Parameter Choice

We explored the space of values for the search parameter γ, which again is the probability with which a rewrite increases the cost by 10% of the baseline. To illustrate the effect of γ, we picked one particular benchmark called *rogers* and ran 100 searches with both cost functions and 8 different values of γ. The results are shown in Figure 5.4. We make several observations: (1) The best value for γ is different for the two cost functions. If we are interested in maximizing the number of fast programs that the search considers (e.g., the number of programs faster than the baseline), then the realtime cost function performs best with $\gamma = 0.0001\%$, while the latency cost function is best at $\gamma = 0.1\%$. Similarly the best choise of γ also varies per benchmark (not shown in Figure 5.4). This validates our experimental setup of first scanning the space of possible values for γ per benchmark and cost function. (2) With $\gamma = 0$, the search never accepts a program with a higher cost, and thus is reduced to hill climbing. Both cost functions return many fewer programs (each search yields significantly fewer than 2000 programs), and especially the realtime cost function finds almost no

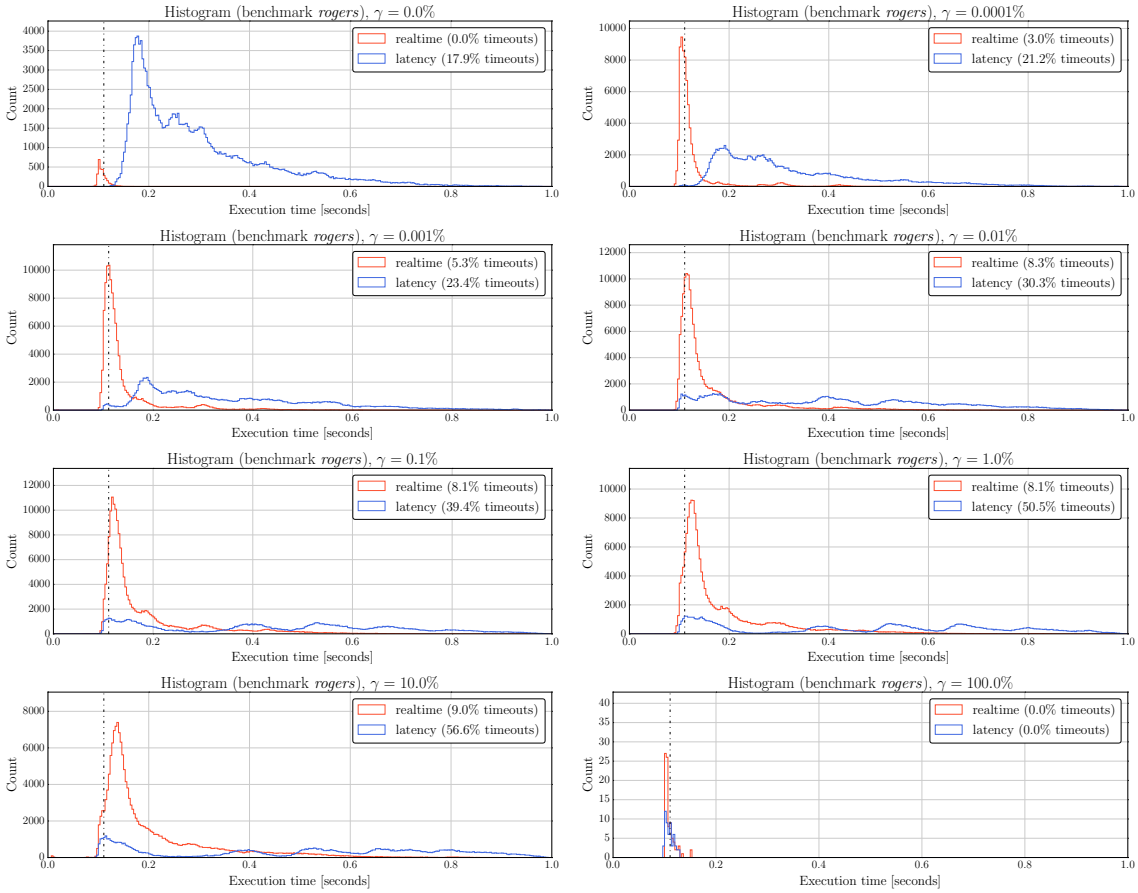

Figure 5.4: Histogram of the 2000 lowest-cost programs of 100 searches with both cost functions and 8 different values of γ. More weight to the left (lower runtime) is better.

programs. We believe the reason is that the realtime cost function is noisy, and so if the search accepts a program with large negative noise, it becomes very difficult to find a lower-cost program later on, as the current cost is lower (due to noise) than the actual cost of the current program. (3) With $\gamma = 100\%$, every program is accepted, even incorrect ones.[4] This quickly leads both cost functions to only consider incorrect proposals, which results in both searches returning almost no correct programs. In practice neither $\gamma = 0$ nor $\gamma = 100\%$ appear to be useful.

5.3.4 Search Space Coverage

To answer (RQ5.2) about the search space that is covered, we sample 2000 programs uniformly at random from all correct programs (i.e., those that pass all the tests) that STOKE explored during the searches, and measure their actual performance. We use a timeout of 1 second. The distribution

[4]The way we restrict searches to only correct programs is through correctness term of the cost function. But for $\gamma = 100\%$ this restriction was effectively lifted and every program was accepted.

of running times is shown in Figure 5.5. For all benchmarks the latency cost function spends a significant fraction of the search on very slow parts of the search space, often several factors slower than even the baseline. For some benchmarks such as *hans*, over 95% of programs time out after 1 second, even though the baseline finishes in 0.101 seconds. The realtime cost function on the other hand focuses its search better on the part of the search space that is relevant for program optimization.

5.3.5 Best Programs Found

In program optimization, at the end of the day we are looking for fast programs, and so the fact that a cost function considers some slow programs during a search may not be an issue. Hence, (RQ5.3) asks about the fastest programs found by both cost functions. In a search of 1 million iterations, STOKE explores 1 million proposals, of which we cannot benchmark all. Even if we only consider the programs that actually pass all tests, we are still left with hundreds of thousands of programs. We do have an estimate of the performance, namely through the cost function. Thus, we collected the 2000 programs with the lowest cost, breaking ties by favoring programs found earlier in the search. We are only interested in programs that are at least as fast as the initial program, so in Figure 5.6 we show for a given threshold improvement (percent decrease in running time over the baseline) on the x-axis how many programs each cost function found.

As mentioned at the beginning of this section, besides the latency and realtime cost functions we also use an additional cost function that only measures correctness and gives no signal about performance. This serves as a baseline for how well pure random search across correct programs works, without taking performance into account. We call this cost function the zero cost function.

We find that the realtime cost function finds faster programs for all 10 benchmarks than both other cost functions, and the latency cost function finds faster programs than the zero cost function in all but 1 benchmark (in which case the difference is only 0.4 percentage points). Looking in more detail at how much better the realtime cost function is over the other two, we find that for 4 of the benchmarks, the difference is significant: 4.9 percentage points for *rogers*, 7.6 for *mentre*, 8.4 for *karim* and 12.5 for *bosmans2*. Furthermore, the realtime cost function finds significantly more fast programs in all cases. Perhaps surprisingly, the latency cost function is within 2 percentage points for the remaining benchmarks, indicating that the random search is often able to find improvements even when the cost function provides relatively little guidance.

The latency-based cost function also clearly provides some useful guidance, as in all but two benchmarks it significantly outperforms the zero cost function. Finally, we note that even a pure random walk on correct programs is able to consistently improve on highly optimized code (recall all benchmarks are compiled with gcc -O3) by a margin of roughly 5%.

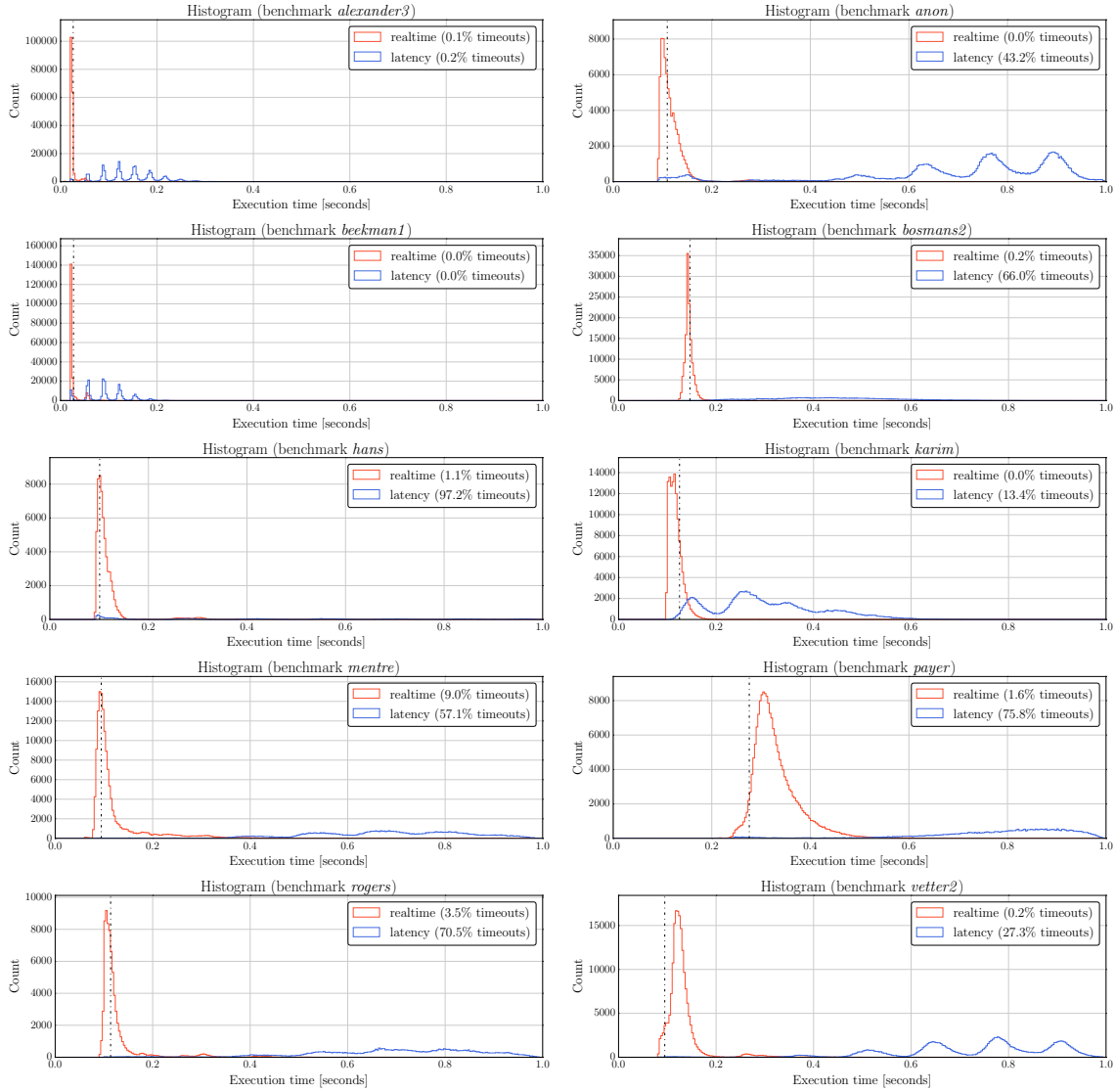

Figure 5.5: Histogram of 2000 randomly sampled programs throughout 100 searches with both cost functions. All programs were benchmarked by plugging them into the original binary. The black dashed vertical line marks the baseline performance of the original program. More weight to the left (lower runtime) and fewer timeouts is better. Programs that took longer than 1 second are not shown (but the frequency of timeouts is indicated in the legend).

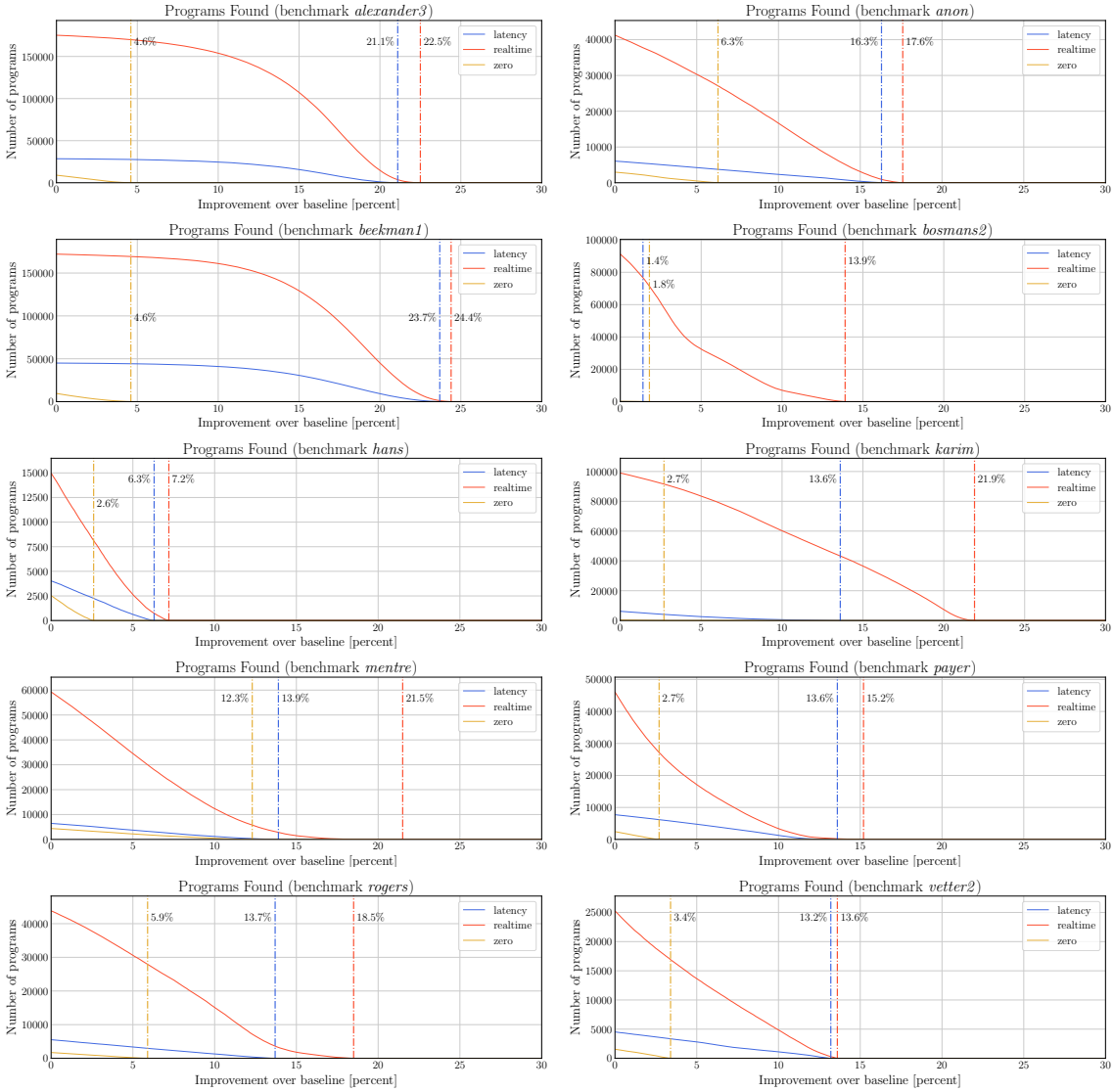

Figure 5.6: Benchmarking the 2000 lowest cost programs of 100 searches with each cost function, showing the number of programs below a given performance threshold (percent improvement over baseline). Higher and more to the right is better.

5.3.6 Search Times

We answer (RQ5.4) by considering the estimated time it takes to find at least one program above a certain improvement over the baseline. We compute this by counting how many of the 100 searches produced at least one program with the given speedup. This gives us an estimated number of searches that need to be performed before the desired improvement is achieved, which we multiply by the average wall-clock time it takes to perform a search. The results are shown in Figure 5.7. These graphs also show the average time a single search takes by considering a 0% improvement over the baseline (since every search by definition finds such a program, thus the estimated number of searches is 1). We cut off the graph on the y-axis when we do not have at least 10 searches of both cost functions succeeding to find at least one program with the given speedup. For such relatively rare events the variation becomes large and we would need to run more searches to gain confidence.

We can see that the time it takes for a single search is roughly comparable for the two cost functions. However, for larger improvements the realtime cost is always at least as fast as the latency cost, and sometimes significantly faster.

5.3.7 Estimator Quality

To answer (RQ5.5), we compute the *Pearson Correlation Coefficient* between the latency cost function and actual performance, as well as between the realtime cost function and actual performance. This is a standard measure of correlation, where 1 (and -1) indicate perfect correlation, and 0 indicates no correlation. We use all programs found by both the realtime and latency cost function to compute the correlation, and we show the results in Table 5.2.

The realtime cost function has quite good correlation, with at least 0.69 for all benchmarks and usually 0.90 or more. The latency cost function on the other hand is a very poor predictor, actually having slightly negative correlation in the majority of cases in the lat-all column. It is important to note, however, that the correlation numbers depend on the set of programs for which the correlation is computed. The lat-all and real-all columns include all the programs found with both cost functions, and since the realtime cost function mostly looks at fast programs, there are many programs with relatively small differences in performance. The latency cost function is very poor in distinguishing these, which is why the correlation is so low. If we restrict the set of programs to just those the latency cost function found (the lat-lat and real-lat columns), then the correlation for the realtime cost function stays about the same, but the latency cost function shows better (and positive) correlation for all programs. This makes sense: the latency cost function can usually distinguish a fast program from a slow program, but is not good at correctly distinguishing two fast programs, particularly if the differences in speed are due to anything other than instruction latencies. The realtime cost function is of course still able to distinguish any performance relevant effects that are not below its measurement noise threshold. In particular, the dramatic difference between the lat-all and lat-lat columns in Table 2 show that the real time cost function finds programs for which

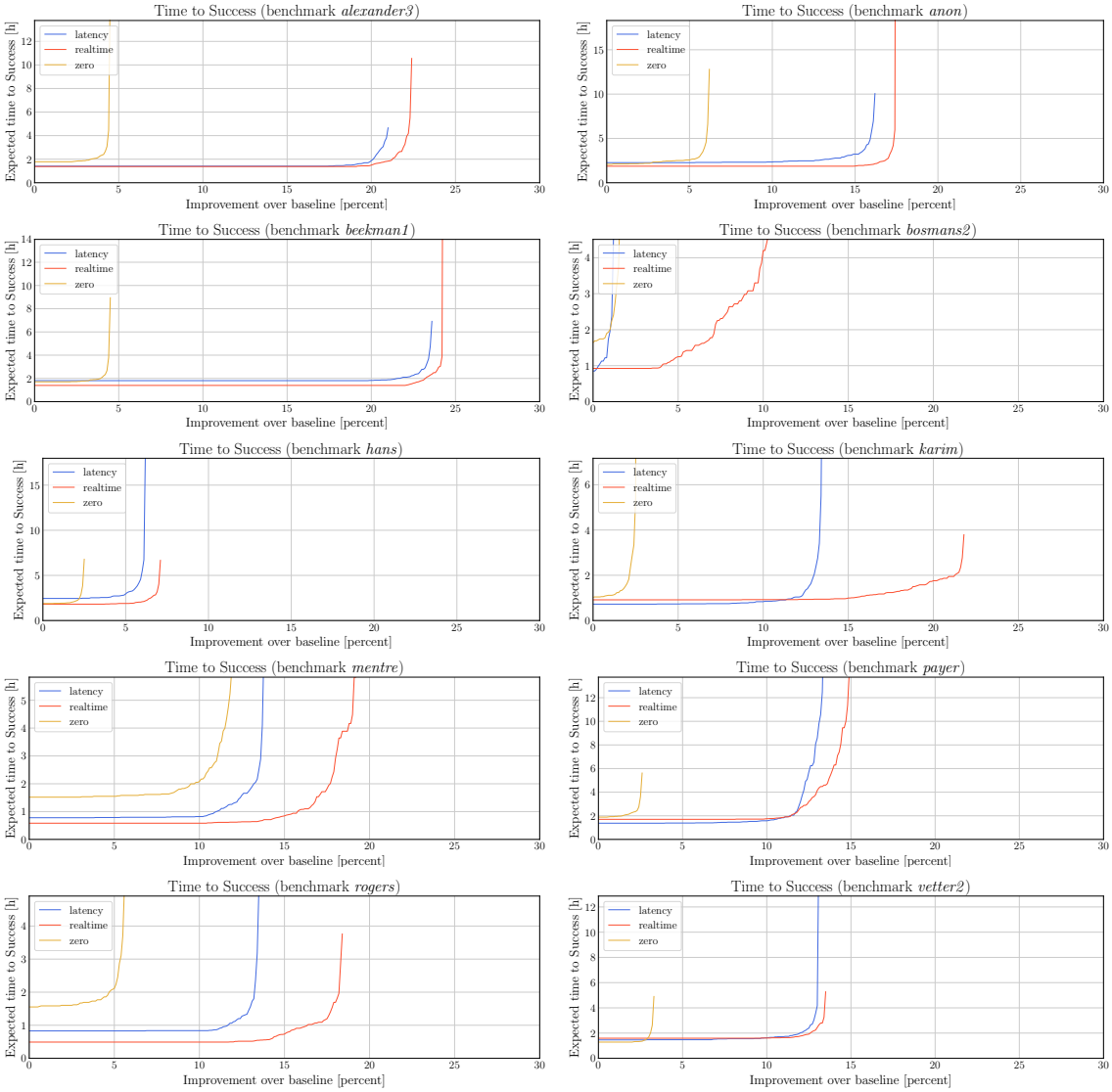

Figure 5.7: Graph of the expected time to find a program with a given speedup, calculated by multiplying the expected number of searches necessary to find a program with the average wall-clock time per search. Lower and more to the right is better.

Benchmark	lat-all	real-all	lat-lat	real-lat
alexander3	-0.17	0.99	0.26	0.97
anon	0.87	0.96	0.81	0.95
beekman1	-0.19	0.97	0.03	0.95
bosmans2	-0.22	0.92	0.21	0.85
hans	0.73	0.96	0.69	0.95
karim	-0.12	0.86	0.42	0.81
mentre	-0.27	0.96	0.19	0.83
payer	-0.10	0.77	0.65	0.86
rogers	-0.22	0.90	0.39	0.69
vetter2	0.67	0.97	0.56	0.93

Table 5.2: Pearson Correlation Coefficient of the latency and realtime cost functions with actual runtime. Closer to 1 is better. The lat-* columns contain coefficients for the latency cost function, and the real-* columns contain coefficients for the realtime cost function. The dataset for the *-all columns are all programs found with both cost functions, whereas the *-lat columns only use programs found with the latency cost function.

the latency cost function is a particularly poor predictor.

5.3.8 Program Optimizations

One might expect that for the benchmarks where the latency and realtime cost function find programs that are almost equally fast, that those programs are roughly the same and both cost functions find roughly the same optimizations. However, this is not the case. To illustrate this and answer (RQ5.6), we calculate the edit distance between the fastest program found with the latency and realtime cost function (edit distance on the list of instructions). This distance is rather large, at 103.8 instructions on average (median of 87 instructions), as Table 5.3 shows. We also calculate the edit distance between the original program (produced with GCC -O3) and the fastest program found with each cost function. These findings emphasize that the optimization is really over the whole program and cannot be understood easily as local changes between programs, even if the performance is close.

5.3.9 Discussion

There are some general observations we can make about the results. First, by all metrics, the realtime cost function performs better than the latency cost function, and sometimes by a wide margin, but the benefit is also variable and sometimes minimal. We believe two factors, both of which are inherent to the problem and not properties of cost functions, can explain the variability.

First, in the unlikely case that the initial program is already the fastest possible implementation of the desired function, no cost function can show an improvement over any other. Thus, at least some of the variability is due to variation in the optimization potential of the different benchmarks. Second, even if a benchmark has significant potential "headroom" for optimization, a sequence of

Benchmark	LOC	baseline-realtime	baseline-latency	baseline-zero	latency-realtime
alexander3	143	147	144	145	154
anon	287	65	258	26	259
beekman1	96	80	79	79	84
bosmans2	56	65	63	64	68
hans	325	87	57	54	89
karim	48	33	34	34	37
mentre	86	80	85	89	87
payer	149	133	133	135	140
rogers	93	80	86	93	87
vetter2	244	33	23	19	33

Table 5.3: Shows the edit distance between the baseline (GCC -O3) program and the fastest found program with every cost function. Also shows the edit distance between the fastest program found using the latency and realtime cost function.

program transformations that produces an optimized program may be more or less common in the search space. Our expectation is that in the two extreme situations where there is either little additional performance to be discovered or the performance improvements that are available are difficult to miss, the difference between better and worse cost functions must be reduced. Conversely, it is in those situations where significant performance gains are possible and finding those gains is non-trivial that better cost functions shine. Our results demonstrate that such situations exist in practice: even for programs that have been written to be high performance and then compiled using production compilers with the highest possible optimization settings, the realtime cost function finds improvements of up to 25% on our benchmarks.

We also show a kind of lower bound on the optimization potential within all benchmarks using the zero cost function, which does not consider performance at all and with respect to performance is a random walk. As one would expect, it performs worse than the other two cost functions in essentially all benchmarks, but is still able to find small improvements by chance; randomly sampling correct programs is an optimization strategy that doesn't work particularly well, but does find improvements nonetheless.

5.4 Limitations and Future Work

This work presents an empirical evaluation of stochastic superoptimization and compares a new proposed cost function with the previously used latency cost function. There are several factors that might limit the results presented as well as areas for potential future research.

- Correctness. We use testcases to evaluate if a candidate rewrite is correct or not. While we believe that our testcases cover the problem well, we do not have a formal guarantee that the rewrites and the target program are in fact equivalent on all possible inputs. Formal verification

is challenging in the presence of loops and arbitrary x86-64 code, but there have been some recent advances that might be applied [CSBA17, SSCA13].

- The set of benchmark programs. While we argue that our set of benchmark programs is rather diverse, using vastly different implementation strategies, instructions and CPU features, our results might not generalize to other programs. We have already observed significant differences in the behavior of stochastic search for different target programs.

- Machine specificity. The optimizations we find are specific to the given machine that the stochastic search was run on, and might not generalize to another machine, especially one with a different CPU. This is by design: this approach attempts to find the fastest possible implementation of a particular program (in the context given by the testcases) when run on a specific machine.

- Transformations. The set of programs that the stochastic search can find is limited by the transformations available to the search. We have not experimented with different transformation strategies, but other work on stochastic search [CSBA17] indicates that this might be a fruitful endeavor.

- Number of iterations. For all our experiments we performed searches of 1 million iterations each.

A couple of directions for future work are:

- Allowing incorrect programs. As described in Section 5.2, we use a cost function that limits the search to programs that pass all testcases. We found that trading off correctness and performance for larger programs (hundreds of lines of assembly) such as the ones in our benchmark set to be challenging. Search would either not consider incorrect programs at all, or get stuck exploring only incorrect programs.

- Investigate what optimizations are found. We have not attempted to categorize the kinds of optimizations that the stochastic search found. In our experience it is challenging and time-consuming to understand a randomly modified assembly program and which of the changes made it faster. This is true for a single rewrite, let alone a whole collection of rewrites.

5.5 Related Work

Exhaustive enumeration. Superoptimization was first proposed by Massalin [Mas87], who exhaustively enumerated short instruction sequences (for Motorola's 68020 CPU), searching for the shortest program that implements a given functionality. Later Bansal and Aiken [BA06] improved the exhaustive enumerative approach by using an SMT solver to prove the equivalence of the short

(up to 3 instructions) rewrites. The cost function they use to estimate performance uses a heuristic to count the number of memory references and instruction latencies.

Stochastic superoptimization. The work most closely related to ours is other projects that use stochastic superoptimization. Schkufza et al. propose the technique in [SSA13], using the latency cost function to estimate performance. The paper shows that for the benchmarks they consider (which were sort straight-line program), the cost function correlates relatively well with actual runtime. However, as we show this correlation appears to be less strong in general, which is why the realtime cost function proposed in this dissertation is able to focus the search more closely on the fast part of the search space.

Other work on stochastic optimization considers different modifications to the cost function by allowing less strict definitions of program equivalence to achieve better performance improvements. For instance, [SSA14] allows lower precision for floating-point programs, while [SSCA15] presents a cost function that assumes certain restrictions on the inputs (such as non-aliasing) to optimize more aggressively. To the best of our knowledge, no other stochastic search technique has explored an alternative performance estimate (not based on latency estimates, or simply the number of instructions).

Symbolic approaches. Several symbolic synthesis and superoptimizers have been proposed [PTBD16, TSTL09, JNR02, CSBA17, JY17, BTGC16, TB13], but it is difficult to reason precisely about the performance of code symbolically. To the best of our knowledge, all symbolic approaches either employ a latency-based cost estimate or simply count instructions.

Performance Evaluation. Curtsinger et al. [CB13] present a methodology to perform statistically sound performance evaluation by sampling from the space of program layouts (stack, heap, code). This is necessary when evaluating an optimization to distinguish it from program layout effects. In our setting, however, we are searching for the fastest program for a particular program layout and particular machine. It is acceptable, even desirable, to exploit layout effects, as we are not looking to apply these same optimizations in any other context.

Chapter 6

Conclusions

Analyzing systems remains an important but challenging aspect of work in the area of programming languages. Historically, to be able to scale to real systems there usually is a need for models that describe certain aspects of a system that cannot be analyzed directly. As systems become larger, more complex and less open, the need for such models will only increase in the future.

In this dissertation, we explored the idea of using guided randomized search to find such models automatically. Guided randomized search, as for example inspired by the Metropolis-Hastings algorithm [MRR+53, Has70], is a good fit for this problem, as it allows for a very flexible cost function that can capture many different aspects of a system and find a model that describes the relevant (and only the relevant) aspects.

With the advent of cloud computing and easy access to lots of parallel computation, guided randomized search is a powerful tool to find models. Not only do the algorithms we proposed scale almost perfectly with more computational resources, the flexibility of the cost function allows for finding higher fidelity models if more computation is available.

Concretely we have shown that guided randomized search is useful in three different contexts. First, we considered the problem of opaque code in JavaScript, which is a significant problem for both static and dynamic program analyzers. We presented a novel technique for synthesizing models for such code by collecting partial execution traces gathered from intercepting memory accesses to shared state. We use a guided random search technique to construct an executable code model that matches these traces. Our technique is implemented in a tool Mimic for JavaScript, and we showed that it could synthesize non-trivial models for a variety of array-manipulating routines.

Second, we targeted the problem of automatically learning a specification of the x86-64 ISA, a very large and complex instruction set. We presented Strata, which is able to learn the semantics of 1,795 x86-64 instruction variants starting from a base set of just 58 instructions and 11 pseudo instruction templates. Strata works by first manually specifying a baseset, and then using guided randomized search to automatically synthesize programs written in just that base set that correspond

to other instructions. We evaluated the approach against handwritten specifications, and found that manually written specifications contain errors (while ours do not), and that our automatically learned formulas are about equally precise and of about the same size.

Third, we tackled another problem with the x86-64 family of processors: program performance estimation for the purpose of program superoptimization. To effectively optimize x86-64 programs, it is important to be able to accurately estimate the performance of a given x86-64 implementation. Previous work on superoptimizers that use guided randomized search used a latency-based estimate, which we show to be a rather poor predictor of performance. Instead, we propose a new cost estimate by directly running the program in bare hardware and show how to overcome some of the challenges in implementing this seemingly simple idea. When we evaluate the new cost function against the previous latency estimate, we find that the new cost function find more fast programs, and finds these programs faster. In several cases the new cost is also able to find significantly faster programs. Perhaps surprising, we also find that the latency estimate is able to find similarly fast programs for some of the benchmarks.

Bibliography

[APV07] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[ATS+15] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 311–327, New York, NY, USA, 2015. ACM.

[BA06] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM.

[BAA15] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 553–566, 2015.

[BBDEL96] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. Rulebase: An industry-oriented formal verification tool. In *Proceedings of the 33rd annual Design Automation Conference*, pages 655–660. ACM, 1996.

[BBP75] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Comput.*, 24(2):122–136, February 1975.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[BEL75] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245,

1975.

[BGRT05] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 250–254, 2005.

[BHJM07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker b last. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, 2011.

[BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. *SIGPLAN Not.*, 51(1):775–788, January 2016.

[CAA15] Lazaro Clapp, Saswat Anand, and Alex Aiken. Modelgen: Mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 129–140, New York, NY, USA, 2015. ACM.

[CB13] Charlie Curtsinger and Emery D Berger. Stabilizer: Statistically sound performance evaluation. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 219–228. ACM, 2013.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[Cha16] Mark Charney. Personal communication, February 2016.

[CKG05] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, volume 31, pages 88–95, 2005.

[CPR07] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Notices*, volume 42, pages 480–491. ACM, 2007.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[CSBA17] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 313–326. ACM, 2017.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248, 2014.

[DRVK14] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. Commutativity race detection. *ACM SIGPLAN Notices*, 49(6):305–315, 2014.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[EHCRI06] César Estébanez, Julio César Hernández-Castro, Arturo Ribagorda, and Pedro Isasi. Evolving hash functions by means of genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1861–1862, New York, NY, USA, 2006. ACM.

[FCD15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[FF00] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *ACM Sigplan Notices*, volume 35, pages 219–232. ACM, 2000.

[FF01] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96. ACM, 2001.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[FF09] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[FNWLG09] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[GHS12] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10, 1967.

[gov] Cpu frequency and voltage scaling code in the linux(tm) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. Accessed: 2017-11-14.

[GRS95] W.R. Gilks, S. Richardson, and D. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC Interdisciplinary Statistics. Taylor & Francis, 1995.

[GT12] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM.

[Has70] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[HG89] JH Holland and D Goldberg. Genetic algorithms in search, optimization and machine learning. *Massachusetts: Addison-Wesley*, 1989.

[HL03] David L Heine and Monica S Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *ACM SIGPLAN Notices*, volume 38, pages 168–181. ACM, 2003.

[HS16] Niranjan Hasabnis and R Sekar. Extracting instruction semantics via symbolic execution of code generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 301–313. ACM, 2016.

[HSC15] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *Foundations of Software Engineering (FSE)*, September 2015.

[HSSA16] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Programming Language Design and Implementation (PLDI)*. ACM, June 2016.

[Int15] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, Revision 325462-057US, December 2015.

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[JMM11] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 59–69, 2011.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002.

[JY08] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th international symposium on Memory management*, pages 131–140. ACM, 2008.

[JY17] Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. 2017.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KHZH06] Hilmi Güneş Kayacik, Malcolm Heywood, and Nur Zincir-Heywood. On evolving buffer overflow attacks using genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1667–1674, New York, NY, USA, 2006. ACM.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming*

Language Design and Implementation, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[KPV03] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[KV08] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 423–427, 2008.

[LDW03] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP 2003), October 23-25, 2003, Sanibel Island, FL, USA*, pages 36–43, 2003.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM, 2006.

[Ler12] Xavier Leroy. The CompCert C Verified Compiler, 2012.

[LP16] Adi Livnat and Christos Papadimitriou. Sex as an algorithm: The theory of evolution under the lens of computation. *Commun. ACM*, 59(11):84–93, October 2016.

[LR13] Junghee Lim and Thomas W. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4, 2013.

[Mas87] Henry Massalin. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987.*, pages 122–126, 1987.

[MLF13] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on*

the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 499–509, 2013.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087–1092, 1953.

[MTT+12] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger sfi for the x86. In *ACM SIGPLAN Notices*, volume 47, pages 395–404. ACM, 2012.

[Neta] Mozilla Developer Network. Array.prototype. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype. Accessed: 2015-03-15.

[Netb] Mozilla Developer Network. Proxy - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed: 2015-03-16.

[nib] Nibble sort programming contest. https://blog.regehr.org/archives/1213. Accessed: 2017-11-14.

[NORV15] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 208–217, 2015.

[OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

[PBG05] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[PH13] Jacques A Pienaar and Robert Hundt. Jswhiz: Static analysis for javascript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE Computer Society, 2013.

[PTBD16] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ACM SIGPLAN Notices*, volume 51, pages 297–310. ACM, 2016.

[PXJ13] Vijay Krishna Palepu, Guoqing (Harry) Xu, and James A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 59–69, 2013.

[QSQ+12] Dawei Qi, William N. Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. Modeling software execution environment. In *Working Conference on Reverse Engineering*, WCRE'12, pages 415–424, Washington, DC, USA, 2012. IEEE Computer Society.

[RB08] Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Compiler Construction*, pages 16–35. Springer, 2008.

[RD06] John Regehr and Usit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*, pages 34–43, 2006.

[RE11] David A. Ramos and Dawson R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification*, 2011.

[RLN98] K Rustan, M Leino, and Greg Nelson. An extended static checker for modula-3. In *International Conference on Compiler Construction*, pages 302–305. Springer, 1998.

[RR04] John Regehr and Alastair Reid. HOIST: a system for automatically deriving static analyzers for embedded systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 133–143, 2004.

[RSY04] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 252–266, 2004.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[SAP+11] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1053–1068, 2011.

[SBY+08] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, pages 1–25, 2008.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[SR15] Venkatesh Srinivasan and Thomas W. Reps. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 596–607, 2015.

[SRBE05] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294, 2005.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013.

[SSA14] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9, 2014.

[SSCA13] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 391–406, 2013.

[SSCA15] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Conditionally correct superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications,*

OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 147–162, 2015.

[SVE+11] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 401–410. ACM, 2011.

[Tan11] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[TB13] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM.

[TLL+10] Aditya V. Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas W. Reps. Directed proof generation for machine code. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 288–305, 2010.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *ACM SIGPLAN Notices*, volume 44, pages 264–276. ACM, 2009.

[WAL] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[WHdM13] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

[WLH+17] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D Ernst, and Zachary Tatlock. Spacesearch: A library for building and verifying solver-aided tools. *Proceedings of the ACM on Programming Languages*, 1(ICFP):25, 2017.

[XA05] Yichen Xie and Alex Aiken. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 115–125. ACM, 2005.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[ZDD13] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 290–306, 2013.