BLACKBOX EQUIVALENCE CHECKING OF PROGRAM OPTIMIZATIONS

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

Berkeley Roshan Churchill

June 2019

Acknowledgements

I would like to especially acknowledge those friends who had a direct influence on the preparation of this thesis. Eric Schkufza was the first one to introduce me to superoptimization and to Alex Aiken's lab at Stanford. In my first few years I learned tremendously from him about organizing a large software project, and also about making nice figures for papers. Rahul Sharma paved the way for much of the work that I did. His work on data-driven equivalence checking opened up a number of questions and new directions for research. His pointers along the way were invaluable. Stefan Heule showed me the value of continuous integration tools, and was a key collaborator on the STOKE codebase, especially when we were getting our first research projects off the ground. Manolis Papadakis was always willing to generously offer his time to answer my C++ questions. Oded Padon played a crucial role in helping refine one of the main techniques presented in Chapter 5 of this thesis – semantic program alignment. Without his assistance, that chapter would have looked very different! Lastly, Alex Aiken played the indispensible role of supporting me throughout the process, and especially helping me learn to focus on what is important. It goes without saying that absent his mentorship there would be no thesis.

It would be impossible to attempt to enumerate all those friends and family members who have been supportive throughout my tenure as a PhD student. To all of you, I offer my sincere love, gratitude and appreciation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developers of safety-critical systems rely on compilers to translate code in a higher-level language, like C or C++, to a low level assembly language. If the compiler incorrectly generates code, it can introduce subtle but serious bugs into the binary. Moreover, these bugs may be excruciatingly difficult to troubleshoot because they belong to a layer of abstraction apart from the typical development environment. While mature compilers exist for many languages and machine architectures today, even these are prone to bugs, particularly when the user supplies non-standard compilation options or when compiling for a less common platform. The majority of these bugs are introduced in optimization passes, in which the compiler applies static analysis techniques to discover optimization opportunities. As a result, safety-conscious users of compilers choose to either accept the risk of compiler bugs, or they may turn off program optimizations to mitigate that risk. Ultimately, we want to offer a third option: compile the code with aggressive optimizations, and formally prove that the optimizations are correct.

Equivalence checking is the problem of formally verifying that two programs, functions or procedures perform the same computation. Equivalence checking may be applied to program binaries, web applications, bytecode and source code. Key applications of equivalence checking include verifying the correctness of compiler optimizations, optimizations performed by hand and software optimizations. Effective techniques for program equivalence checking also enable speculative optimization techniques, such as superoptimization. Of particular interest is *black-box* equivalence checking, wherein the equivalence checker operates only on the low-level code, and does not have access to typing information nor knowledge of the program analyses and optimization passes performed by the compiler.

In this thesis, we use black-box equivalence checking to verify the correctness of optimized `x86-64` code generated by compilers and superoptimizers. There are three main ways this thesis expands the applicability of equivalence checking:

- we demonstrate how a method called *alias relationship mining* can be used to generate efficient SMT queries for checking equivalence in the presence of memory aliasing (Chapter 3);

- We advance the state of the art of loop superoptimization by showing that a sound superoptimizer depends on both sound equivalence checking and bounded equivalence checking (Chapter 4); and
- we introduce a new and powerful technique, called *semantic program alignment*, that allows checking the correctness of aggressive optimizations that alter control flow (Chapter 5).

The next section describes these contributions in greater detail.

1.1 Contributions

One of the core difficulties, and often the most expensive part, of verifying properties of low-level code is reasoning about the program's use of memory. Chapter 3 tackles the problem of generating SMT constraints to represent memory accesses. To model memory accesses to the heap soundly, one must consider the possibility that two memory accesses may *alias*. In general, if there are n memory dereferences in a program, there is no a priori information about whether or not these dereferences overlap with one another. In the general equivalence checking setting, one needs to consider every possible combination of overlaps. This task may be offloaded to the SMT solver using the theory of arrays, and often this solution is adequate, but sometimes the SMT solver becomes stuck exploring the space of possible memory reference overlaps too slowly. We introduce a technique, called *alias relationship mining* (ARM) where we use test cases to guess and prove aliasing relationships between the memory dereferences. ARM improves the scalability and predictability of the equivalence checking routine; while guessing and proving the aliasing relationships has a cost, it dramatically reduces the search space of aliasing configurations that the SMT solver may need to check, which reduces the likelihood that the SMT query is intractable. In Chapter 3, we describe how ARM works, but it is only fully evaluated in the context of experiments presented in Chapters 4 and 5.

Superoptimization is a speculative optimization technique wherein an optimizer searches for an optimal program within a search space. This is in contrast to traditional optimization techniques which depend on a pre-selected set of optimizations programmed into a compiler. Not only are compilers limited to a fixed set of optimizations, but they often suffer from the *phase ordering problem*, in which the order of optimizations performed can impact the performance of generated code; superoptimizers avoid the phase ordering problem because they are agnostic to any ordering of transformations. Chapter 4 focuses on fully-verified loop superoptimization for realistic benchmarks. Past work on superoptimization generally focused on loop-free code or did not provide formal guarantees of correctness. Previously, only one work [59] demonstrated the viability of performing formally verified superoptimization for loops. However, that work only scaled to functions of up to 10 lines of assembly code. The key scalability problem was that the superoptimizer did not have enough test cases to guide it to correct and fast programs, and as a result the equivalence checker would reject the optimizations. We demonstrate that a *bounded equivalence checker* can be used as an intermediary between the superoptimizer and the formal equivalence checker to guide the superoptimizer to a correct optimized program. We demonstrate this technique specifically in the context of *Google Native Client* (`NaCl`), a software fault isolation system; generating high performance `NaCl` code is a hard optimization problem, and we show that our superoptimization technique is capable of obtaining speedups over production code shipped by Google. The contributions in Chapter 4 include:

- A demonstration that stochastic superoptimization has a significant advantage over conventional compiler technology for Google Native client; we demonstrate a median speedup of 25% across 13 `libc` functions shipped with `NaCl` by Google. Our optimized binaries may be used as a drop-in replacement and are backed with a guarantee of formal equivalence to the original code.

- A new and robust architecture for stochastic program search over code containing loops. Our approach combines a bounded equivalence checker and a sound equivalence checker for proving loop equivalence. This is the first application of stochastic superoptimization to a real-world domain of programs with loops.

Semantic program alignment offers a powerful extension of equivalence checking technology to a new class of benchmarks. Prior work on equivalence checking in a black-box setting has been unable to prove the correctness of compiler optimizations in the presence of unforeseen control flow optimizations on loops, such as vectorization, loop unrolling and loop peeling. The main issue has been that of *aligning* executions of the two programs; in the past, many works have assumed a simple one-to-one correspondence between the iterations of the loops of one program and the loops of the other. We offer a general technique for aligning two programs using test data, and demonstrate that this alignment technique is suitable for challenging equivalence checking problems out of reach of prior work, most notably including a formal proof that the handwritten vectorized `strlen` implementation that ships with `libc` matches a reference version. Semantic program alignment is presented in Chapter 5; the contributions of that chapter include:

- A novel and robust approach for semantics-driven construction of product programs using alignment predicates and trace alignments.

- A set of 56 realistic `x86-64` benchmarks for evaluating equivalence checking techniques on optimizations that alter control flow, such as loop unrolling, loop peeling and vectorization.

- The first fully automatic black-box algorithm for proving the correctness of vectorization optimizations as performed by modern compilers on `x86-64`.

- A demonstration of a useful, real-world application of our equivalence checking technology to verify the correctness of a handwritten vectorized implementation of the `strlen` function shipped in `libc`.

Chapter 2

Background and Related Work

2.1 Inductive Proof of Equivalence

Proof by induction is a basic technique used in many equivalence checking works. In the simple case where the two programs each have one loop and the loops execute the same number of times, induction may be performed on the number of loop executions. The key is to use a *relational invariant*, $\mathcal{I}$, which relates the states of the two programs at each loop iteration; this forms the inductive hypothesis. There are a few verification conditions, each of which make a statement about a loop-free fragment of code; these can be discharged with an SMT solver.

- For a given input, both programs enter the loop body or both programs exit.
- If the loop body is entered, $\mathcal{I}$ must hold on the first iteration.
- If $\mathcal{I}$ holds after n loop iterations, then (i) both programs exit, or both programs go on to the $n + 1$st loop iteration; (ii) if both programs continue, $\mathcal{I}$ holds on the $n + 1$st loop iteration; and (iii) if both programs exit, then the output values are provably equivalent.

This technique generalizes to scenarios involving more loops or nested loops when the correspondence between loop executions between the two programs is still clear. One starts by selecting a set of *cutpoints*, where a cutpoint is a pair of program points – one from each program [59, 7]. One cutpoint is necessarily the entry point of both programs, and another cutpoint corresponds to the exits. There must additionally be a cutpoint in each loop. A simple approach is to prove that the two programs operate in “lockstep”, meaning that, if both programs execute from the entry, then they both transition to the same cutpoint. In practice, this means that both programs enter a corresponding loop, or they both exit. The inductive step involves showing that if both programs start at some cutpoint $\mathcal{C}$, then they both transition to the same cutpoint $\mathcal{C}'$. A relational invariant is chosen at each cutpoint that relates the states of the two programs, and these relational invariants are used as the inductive hypothesis.

```
int f(int x, int n) {
  // cutpoint A
  int i, k = 0;
  for (i = 0; i != n; ++i) {
    x += k*5;
    k += 1;
    if (i >= 5)
      k += 3;
    // cutpoint B
  }
  return x;
  // cutpoint C
}
```

```
int f'(int x, int n) {
  // cutpoint A
  int i, k = 0;
  for (i = 0; i != n; ++i) {
    x += k;
    k += 5;
    if (i >= 5)
      k += 15;
    // cutpoint B
  }
  return x;
  // cutpoint C
}
```

Figure 2.1: A simple equivalence checking example [59]

Consider the example pictured in Figure 2.1 of an equivalence checking problem arising from a strength reduction optimization. The functions f and f' are given three cutpoints. Cutpoint A corresponds to program entries, cutpoint B is in the loop bodies, and cutpoint C relates the exits. Notationally, if v is a variable in f, we use v' to denote the same variable in f'. At cutpoint A we assume the input values are equal, that is $\phi_A = \{x = x' \wedge n = n'\}$. At cutpoint B, one may guess or infer the invariant $\phi_B = \{x' = x \wedge k' = 5k\}$ (see Section 2.3). At cutpoint C we hope to prove $\phi_C = \{x = x'\}$. The following proof obligations must be checked to guarantee equivalence and these also may be discharged with an SMT solver:

- From cutpoint A, function f takes the branch to cutpoint C (short-circuiting the loop) if and only if f' also does.
- If both f and f' short-circuit the loop, then both terminate and ϕ_C holds.
- If f and f' enter the loop and reach cutpoint B, then the invariant ϕ_B holds.
- From cutpoint B, function f takes another loop iteration if and only if f' also does.

- If f and f' execute from cutpoint B in states satisfying ϕ_B and execute another loop iteration, then ϕ_B holds when both functions next reach cutpoint B.
- If f and f' execute from cutpoint B in states satisfying ϕ_B and both functions exit without another iteration, then ϕ_C holds.

An alternative formulation of this technique is to compose the two programs into one *product program*. Each execution of the product program corresponds to an execution of both starting programs. Here, we can perform equivalence checking (or checking of other relational properties) by using standard single-program verification techniques on the product program. We learn invariants for each loop, and prove by induction that they always hold. Then, we prove assertions which imply that the two programs have the same outputs and that the loop executions of the product program are in one-to-one correspondence with f and f'.

For the example in Figure 2.1, one possible product program is depicted in Figure 2.2. Here we have renamed all the variables of f' so that they do not clash with those of f, and have placed the statements in the loops of each of f and f' into one loop. We add assert statements to check that f and f' take the same branches as each other, and assert that the values of x and x' are equal at the program exit. Equivalence can be established by proving that the assert statements always fold. We have thus reduced the equivalence checking problem to verifying assertions of a single program, and the standard techniques (e.g. proof by induction with a loop invariant) apply.

In the following sections, we see how different works approach the question of identifying cutpoints and invariants, or constructing product programs. Translation validation (Section 2.2) uses either static techniques or instrumentation of the compiler; data-driven techniques use test cases provided by a user (Section 2.3); and some techniques use manually provided information.

```
void ff'(int x, int x', int n, int n') {
  assume(x == x' && n == n');
  int i, k, i', k' = 0;
  for(i = 0, i'=0; i != n && i' != n'; ++i, ++i') {
    assert((i != n) ^ (i' != n') == 0);
    x += k*5; x' += k;
    x += 1;   k' += 5;
    if ( i >= 5 )
      k += 3;
    if ( i' >= 5 )
      k' += 15;
  }
  assert(x == x');
}
```

Figure 2.2: Product program for the example

.

2.2 Translation Validation

Translation validation [45, 50, 61, 57] verifies that optimization passes performed by a compiler preserve program semantics. Often – but not necessarily – translation validation is performed on intermediate representations after each optimization pass, and has access to type information or other static analysis performed by the compiler that may be difficult or impossible to derive from the assembly code in a black-box setting [52]. Necula uses symbolic execution to learn the simulation relation between two programs [45]. The technique requires that the compiler does not introduce new branches into the program; as a result, this work cannot handle some compiler optimizations, such as loop unrolling. By exploiting the correspondence between branches, it is possible to identify pairs of program points (analogous to the cutpoints described above) along with relational invariants which form a simulation relation, which is leveraged for an inductive proof of equivalence. In this work, translation validation can be performed without any instrumentation of the compiler. More recent black-box equivalence checking work, such as [17], can be thought of extending

Necula's work with more general control flow.

Other works extend translation verification to loop optimizations using different techniques. The authors of [71] add instrumentation to the compiler to generate anticipated invariants for each optimization, and then these invariants are used to derive verification conditions. Additionally, they consider how to deal with transformations like loop tiling and loop unrolling that reorder execution. In [7] the authors detect and prove the correctness of transformations that reorder loop iterations with less information from the compiler, before proceeding to a second phase where cutpoints are chosen and a dataflow analysis is used to infer loop invariants.

Some works take an entirely different approach to translation verification. In [36] the authors instrument the compiler to generate a trace of program optimizations performed, and each class of transformation is verified. A pre-defined set of verification conditions for each type of transformation is built into the tool, so it requires non-trivial work to support new optimizations. Equality saturation [62, 64] builds graphs of expressions for each program, transforms the graphs via a series of rewrite rules, and checks for equality. These approaches depend on the manually specified rewrite rules and do not attempt to build a product program.

2.3 Data-Driven Invariant Inference

Daikon [46, 24] was the first dynamic invariant learning system used for verification of single programs. The authors observe that guessing invariants from execution traces, and formally checking them later, can be more efficient than guessing invariants without any kind of dynamic analysis (as in [27]) or trying to statically analyze a program to learn invariants. There has been extensive follow-up work outside the context of equivalence checking [15, 58].

In [59] the authors offer a new way to dynamically learn invariants for equivalence checking. The user provides a set of test cases to guide verification. Such test cases may be generated randomly, by bounded techniques, or written manually by developers. Given the test data, the two programs are run on each test case. At each cutpoint, a set of pairs of machine states (σ, σ') are collected. The goal is to learn a relational invariant I such that I holds for each (σ, σ') pair, while avoiding extremes of overfitting and underfitting.

The following approach is shown in [59] to be particularly effective for invariants in the class of linear equalities: given M variables from both programs at a cutpoint and N pairs of data, one constructs an $N \times M$ matrix where the ith row corresponds to one (σ, σ') pair, and the jth column corresponds to one variable or register. The nullspace of this matrix is computed over the field of rational numbers, and each basis vector in the nullspace corresponds to a linear equality over the M variables that holds over the data. Moreover, since the basis vectors span the entire nullspace, the conjunction of linear equalities thus learned actually imply all linear equalities over the data. (See [28] for background on linear algebra.)

These data-driven inference techniques are not sound, of course, but offer candidate invariants that may be used for subsequent verification. In some works, the invariants learned from dynamic or static techniques need to be exactly correct to succeed in verification. In others, it is sufficient to provide a set of candidate invariants. For example, Houdini [27] introduces a technique where number of invariants $I_1, I_2, \ldots, I_n$ are guessed with the expectation that some hold and some do not. Then, one attempts to prove equivalence using the conjunction $I_1 \wedge \cdots \wedge I_n$ as the invariant. Some conjuncts will cause the proof to fail; these conjuncts are eliminated, and the proof attempt is repeated. Eventually a fixed point will be reached where the proof succeeds, or in the case that the set of invariants is insufficient, all conjuncts are eventually discarded. Other works use symbolic execution to refine invariants learned by dynamic techniques [15, 69].

2.4 Other Equivalence Checking Techniques

Dahiya [17] offers a novel black-box equivalence checking algorithm for compiler optimizations. They construct and check a product program called a joint transfer function graph (JTFG) that aligns execution paths in the two programs for verification. There are a few limitations, however. First, they require that branch conditions of one program provably match branch conditions of the other. Second, they assume that paths in one program correspond to sets of paths in the other, rather than allowing a many-to-many relationship. Lastly, their algorithm does not scale well in the case of loop unrolling and vectorization. In Chapter 5, we show how our techniques overcome these limitations using a different approach to program alignment.

In [19, 20] authors use constrained Horn clauses to summarize the entire execution of two programs and then use Horn clause solvers to prove equivalence. To make the problem tractable, the authors introduce a transformation called predicate pairing, where predicates from the two programs are combined into one predicate that models both programs. These predicates usually correspond to matching one iteration of a loop or one recursive step in one program with one in the other. The authors do not report on their ability to handle optimizations such as vectorization or loop unrolling where loop iterations or recursive steps are executed in one program more often than the other.

Some works use manually provided specifications to perform alignment. For example, in [38] the authors prove the correctness of some difficult compiler optimizations, such as loop interchange, by using programmer-supplied templates which imply a correspondence between loop-free code fragments. Other works allow users to specify “control flow synchronization points” manually [37].

Polycheck [5] dynamically verifies the correctness of transformations of affine programs that reorder iterations, such as tiling and sectioning. It uses a modified compiler to generate an instrumented binary that performs checks at run-time.

In [22] authors verify loop parallelization and vectorization optimizations, but are limited to cases where the control flow of the program remains the same for all possible inputs, which excludes many practical situations. Other recent work considers the correctness of peephole optimizations without loops [40], where authors make progress on problems such as modeling undefined behavior.

2.5 Bounded Equivalence Checking

Bounded equivalence checking techniques have an altogether different goal – we describe it here due to its relevance in Chapter 4 of this thesis. Instead of a sound guarantee that two programs are equivalent, a bounded equivalence checker will verify that all pairs of executions up to a certain length are equivalent. This is useful for bug-finding; a bounded equivalence checker can determine if a short execution of both programs exists that exhibits a semantic difference between them. Moreover, most bounded techniques generate a counterexample that can be used to reproduce the differing behavior. This technique is developed in SymDiff [31], differential symbolic execution [49] and Currie's work [16].

UC-KLEE [51] applies this technique to a real-world setting, where it is used for the relational verification problem of checking that patches do not introduce security vulnerabilities. The key problem with prior works is the exponential blowup in paths that the bounded equivalence checker must consider. As a result, applying the technique to whole programs is intractable. UC-KLEE instead performs under-constrained equivalence checking, wherein individual functions may be compared with each other. This increases the risk of false-positives, which the authors mitigate through a number of heuristics.

2.6 Modeling Memory

All equivalence checkers, including bounded and unbounded checkers, need to encode semantics for memory accesses into the SMT queries. The trouble is that, a priori, any two memory accesses may or may not refer to the same location. In some cases two memory accesses provably alias each other or provably access distinct memory locations, and without this information an equivalence checker may be overly conservative. Given N potentially aliasing memory accesses, each of size M, the number of different ways that they can overlap is exponential in N, and also grows with respect to M as well, so for large choices of N and M, considering each case may be intractable (as seen in [59]).

There are a few approaches to this problem in prior work. An unsound technique, which has been successfully used in [51] and other works, is to simply assume that pointers which are inputs to a function do not alias. One sound technique is utilizing an SMT solver that supports the theory of arrays. The program semantics are encoded by updating and reading from a series of arrays, where each array represents a snapshot of the heap state. Such tools are said to use a *flat memory model* since the heap (and sometimes the stack too) are modeled as a single homogeneous key-value store. A major advantage of this technique is simplicity, as it requires no separate program analysis. The disadvantage is that the SMT solver must reason about all the aliasing cases on its own; sometimes the SMT solver may do so efficiently, but the solvers do not perform predictably or consistently. In the worst case, the solver may traverse an intractably large number of cases and time out.

A refinement of the above technique is to partition memory accesses into sets where accesses from set i cannot alias with accesses from set j [66]. Then, a separate array can be used for each set of memory dereferences. The partitioning can be done, for example, by applying a pointer analysis. In the *Burstall model* [11], memory accesses are partitioned by types, with a separate heap for each different type used in the program. For example, 4-byte integers may be stored in one heap, pointers to strings in another, and so on. This is sound when the types of all the variables are known statically, and the language can guarantee that references to different types do not alias. Using separate heaps for each type reduces the number of cases to consider – sometimes dramatically. However, on `x86-64` there is no typing information from a source language available for us, and in practice we frequently find memory reads and writes of different sizes that alias (for example in vectorization optimizations). Similarly, one may use a separate array to model the stack and the heap; this can be done in situations where memory dereferences provably reference the stack or the heap but not both.

Value Set Analysis (VSA) is a flow-sensitive, context-sensitive interprocedural analysis that operates on low-level code containing pointers in the absence of types. VSA tracks the set of integer values in the program using a strided interval domain. There will naturally be cases where the abstraction works well and provides exactly the desired information, and other cases where the overapproximated sets insufficiently constrain the aliasing relationships [4]. A disadvantage of VSA is that it requires implementing some kind of abstract semantics for the entire instruction set. Follow-up work suggests other abstract domains and an intermediate language to simplify implementation of VSA [70].

2.7 Superoptimization

A motivating application for equivalence checking is *superoptimization*. A superoptimizer optimizes a program X by searching for a faster program Y (in a potentially large search space) that performs the same computation as X. The seminal work on superoptimization by Massalin [44] simply tests that two code fragments generate the same outputs when run on a large number of inputs. Likewise, more recent works have skipped formal verification, especially in the domain of floating point codes where verification is more difficult [54]. STOKE [53, 55] and other superoptimizers [42, 35, 29] improve on this by using an SMT solver to check equivalence for straight-line code. DDEC [59] was the first use of an equivalence checker to verify superoptimized code containing loops – but this was limited to only small examples of less than ten lines of assembly code. In Chapter 4, we show how equivalence checking enables superoptimization of loops in a real setting.

Chapter 3

Alias Relationship Mining

A major problem for sound equivalence checkers is the exponential growth in the number of ways in which memory dereferences may alias. In general, every pair of memory references must be compared, and for each pair of references, there are often several cases; the two references might be non-aliasing, but if they do alias and are both larger than one byte in size (the smallest addressable unit on `x86-64`), then there are several ways that the two references may overlap. The number of combinations grows rapidly with the number and size of memory accesses, and every possible case must be considered. Past authors use source code information, static analysis, or offload the problem to the SMT solver. However, a black-box equivalence checker does not have access to source code information, and implementing a program analysis like VSA [4] with sufficient accuracy can be prohibitively time-consuming. We instead introduce *alias relationship mining* (ARM), a new technique which uses test data to soundly reduce the number of aliasing cases by several orders of magnitude. ARM allows equivalence checking techniques to support a broader class of benchmarks reliably, without encountering timeouts when checking proof obligations.

The basic idea behind ARM is that one can discover which memory locations alias by executing the code and monitoring the execution. In many cases, a single execution or a few executions are representative of the aliasing behavior of all executions, and by observing such executions we can form general hypotheses about which memory accesses can alias, and what overlap exists (if any). We then use an SMT solver to check these hypotheses. Whichever hypotheses are shown to hold for all executions are used to reduce the number of cases that need to be considered to prove equivalence.

ARM is well suited to data-driven systems such as superoptimizers where test cases are already available. There is a great deal of existing work in the symbolic execution, program analysis, and bounded model checking communities on memory models (see Section 2.6), and there are other approaches to avoiding worst-case behavior in analyzing aliasing. One advantage of ARM is that there is no need to

```
# f                                 # g

movl esi, esi                       movl esi, esi
movl (r15,rsi), edx #a              movl (r15,rsi), edx  #a'
movl edi, eax                       addl 4, esi
addl 4, esi                         nop (23)
movl edi, edi                       movl edi, eax
movl edx, (r15,rdi) #b              movl edi, edi
addl 4, edi                         movl edx, (r15, rdi) #b'
testl edx, edx                      shrl 1, edx
movl esi, esi                       movl esi, esi
movl (r15,rsi), edx #c              movl (r15,rsi), edx  #c'
addl 4, esi                         addl 4, esi
movl edi, edi                       movl edi, eax
movl edx, (r15,rdi) #d              movl edi, edi
addl 4, edi                         movl edx, (r15, rdi) #d'
testl edx, edx                      shrl 1, edx
retq                                retq
```

Figure 3.1: Two unrolled functions that perform a wide-string copy for input strings of length 2 (including null-terminator). The code is ATT syntax with `%` and `$` prefixes removed for space. Accesses a, a' read the first source character; b, b' write this character into the destination; c, c' read the second (null) character; and finally d, d' write the null character into the destination.

write separate semantics for `x86-64` instructions, like those which would be necessary to implement a separate pointer analysis.

In this chapter we offer only a brief introduction to ARM and a minimal experiment to demonstrate its utility. Both Chapters 4 and 5 use ARM as part of the implementation of equivalence checkers, and only in the context of those chapters is ARM truly evaluated.

3.1 Motivating Example

Figure 3.1 shows two loop-free fragments of `x86-64` code for which we wish to check equivalence. These loop-free fragments arise from unrolling generated code from the `wcpcpy` benchmark, which will be discussed further in Section 4.2. These two fragments perform a string copy on 4-byte wide character strings, where the input has only one non-null character followed by a null terminating character. This code contains eight memory accesses in total, four in f (labeled a, b, c, d) and four in g (labeled a', b', c', d'). In this example, the accesses are in one-to-one correspondence; when executed, the addresses dereferenced by f are always the same as those dereferenced by g. Moreover, accesses a and c always refer to the consecutive 4-byte wide characters in the source string; similarly b and d refer to consecutive characters in the destination.

It would be tempting, but incorrect, to model the memory with four 4-byte non-overlapping pseudo-registers, one for each pair of corresponding accesses. The problem is that the source and destination strings may overlap, which happens when the initial state satisfies $\texttt{\%rsi} - \texttt{\%rdi} = \epsilon$, for $-8 < \epsilon < 8$. Thus, we must consider 15 cases, one for each value of ϵ, and then one more when the strings do not overlap. Each case corresponds to an *aliasing configuration*, which is a description of how the memory accesses overlap. For a given aliasing configuration, we can model the memory as a set of pseudo-registers. Therefore, to prove equivalence of f and g, we must give a total of 16 queries to the SMT solver, one for each possible aliasing configuration.

Some works model memory by enumerating all possible aliasing configurations, which very quickly becomes intractable: There are over 50 million aliasing configurations for eight 4-byte memory accesses, if we do not use the relations described above (e.g. a, a' alias; a, c are consecutive; etc.). This approach makes verification infeasible for all but the smallest examples. For example, the DDEC validator in [59] took two hours to validate two assembly sequences with less than 10 LOC and two dereferences each.

In alias relationship mining, we run f and g on a set of concrete test cases to learn these aliasing relationships. Let $A(\mu)$ denote the symbolic address of memory access μ. In our example, we learn the following from concrete data: $A(a) = A(a'),\ A(b) = A(b'),\ A(c) = A(c'),\ A(d) = A(d'),\ A(c) = A(a) + 4,\ A(d) = A(b) + 4$. We verify these relationships by translating them into bitvector formulas and verifying their validity with the SMT solver. For example, $A(a) = \%\texttt{r15} + \%\texttt{rsi}$ and $A(c) = \%\texttt{r15} + \%\texttt{rsi} + 4$, so to verify that $A(c) = A(a) + 4$, we query the SMT solver with $\%\texttt{r15} + \%\texttt{rsi} + 4 \neq (\%\texttt{r15} + \%\texttt{rsi}) + 4$; when the solver reports "unsat", this proves the relationship holds. We use the verified relationships to model the memory state with pseudo-registers. In this example, we use two: an 8-byte register for accesses a, a', c, c' and another for b, b', d, d'. These two pseudo-registers may or may not overlap; we have to invoke the SMT solver a total of 16 times, one for each possible aliasing configuration.

3.2 Technique

Given f and g, our goal is to model every memory access as a read or write to a set of pseudo-registers. We assume that f, g are unrolled loop-free code, so enumerating all memory dereferences is trivial. The problem is that we do not know if two accesses will reference the same memory location, different locations, or if they will partially overlap. We use test cases $x_1, \ldots, x_n$ to learn *aliasing relationships* between the different accesses. Usually $n = 1$ is sufficient. Let $A(\mu)$ denote the symbolic representation of the address dereferenced by access μ; we derive $A(\mu)$ through symbolic execution of f and g. For a pair of accesses μ and ν, an aliasing relationship is a statement of the form $A(\mu) - A(\nu) = \epsilon$, where ϵ is an integer constant.

Let $A_j(\mu)$ denote the concrete address of memory access μ when f or g is run on test case x_j. For each pair of memory accesses, μ and ν, we check if the concrete values $\epsilon_j = A_j(\mu) - A_j(\nu)$ are constant for all j. If so, then we infer the aliasing relationship $A(\mu) - A(\nu) = \epsilon$. We then use the SMT solver to check if this statement is valid.

Once finished, we have a set of verified relationships of the form $A(\mu_i^*) - A(\nu_i^*) = \epsilon_i^*$. We place the memory accesses into equivalence classes, where the related accesses μ_i^* and ν_i^* are in the same class. Memory accesses in the same class are at a fixed offset to each other. We can model the memory used by all the accesses of one class with a fixed-size pseudo-register. Each access corresponds to reading or writing a sub-range of this pseudo-register. These pseudo-registers may overlap; we explicitly enumerate all the ways they may do so, and invoke the SMT solver once for each aliasing configuration.

3.2.1 Alias Relationship Mining Without Test Cases

In the absence of test cases, one cannot predict the difference $A(\mu) - A(\nu)$ simply by inspecting traces of executions. There are two remedies for this problem – both of which we have implemented.

One approach, which is limited in scope, is to use a domain-specific heuristic to guess a superset of relationships of the form $A(\mu) - A(\nu) = \epsilon$ that may hold and then use the SMT solver to check them. In our superoptimization work on string benchmarks (see Chapter 4), we were able to guess that memory dereferences are likely to be adjacent. For example, if a benchmark iterates over a string of 4-byte characters, then we would expect that there exist accesses μ and ν such that $A(\mu) - A(\nu) = 4$. By testing for this relationship for all pairs of accesses, we can find all the equivalence classes described above.

A more general approach is to dynamically generate more test cases with an SMT solver, and then use the execution traces to learn the value for ϵ. This works well in practice, and the queries to the SMT solver are simple. In the context of a whole-program equivalence checker, we typically only need the test case to satisfy the invariant relating the execution states of f and g before the two code fragments are run. We implemented this approach for our semantic program alignment system (see Chapter 5).

3.3 Implementation

Correctly and efficiently enumerating all the ways that different equivalence classes may overlap is easy when there are only two classes to consider. However, with three or more equivalence classes, a good implementation becomes quite tricky. Moreover, invoking the SMT solver repeatedly with very similar problems can sometimes put the solver at a disadvantage. This problem can be remedied by using the push and pop features of SMT solvers, but this fix makes constraint generation even trickier to implement. There are a few ways to simplify the implementation, however.

First, one can attempt to prove that memory dereferences in different equivalence classes never overlap. If this property holds, then there is only one case for the SMT solver to check. To establish this property, one can attempt to prove an overlap is impossible for each pair of memory dereferences belonging to different equivalence classes. However, this is unnecessarily expensive; an optimization is to coalesce adjacent memory dereferences in the same equivalence class first.

A second approach is to utilize the theory of arrays. Suppose there are m equivalence classes of size $s_1, s_2, \ldots, s_m$. To generate constraints, we introduce a bitvector variable X_i to represent the address of the ith equivalence class. An array A represents the state of the heap. When generating a constraint for an instruction with a memory access in class j, we use the expression $A[X_j + k]$ to reference the memory. The advantage over the flat model can be understood by considering the inner workings of the SMT solver.

SMT solvers utilizing the theory of arrays need to use other theories to prove whether two array indices are equal or not – and, in the worst case, this may be needed for every pair of array indices. Usually only the theory of bitvectors is necessary, but so too may be the theory of uninterpreted functions if benchmarks use opaque functions to address memory. In the general case, performing this computation might be very difficult, depending on the actual array indices. However, with our technique, we know exactly how difficult it is to determine the (dis)equality of array indices: for constants k_1, k_2, the validity of $X_i + k_1 = X_i + k_2$ or $X_i + k_1 \neq X_i + k_2$ can be determined by unit propagation alone if transformed into a boolean satisfiability problem. Comparing $X_i + k_1$ and $X_j + k_2$ will be harder, but this is indeed the necessary case-work; if the solver introduces the clause $X_i + k_1 = X_j + k_2$, this corresponds to exactly the aliasing configurations where $X_i - X_j = k_2 - k_1$, and the unit propagation will then be sufficient to find the remaining equalities and disequalities for the theory of arrays to reason about the equivalence classes indexed by X_i and X_j. The SMT solver's performance ultimately depends on the heuristics and implementation details within the solver, but this implementation technique helps increase the likelihood that the solver searches the space of partial satisfying assignments efficiently while keeping the implementation simple.

This technique was implemented for our experiments in Chapter 5, and we show that it is helpful in proving the correctness of benchmarks where the flat memory model fails. However, we do not evaluate this technique in depth or directly compare it to a brute-force enumeration of aliasing configurations outside the SMT solver.

3.4 Evaluation

The key difference between alias relationship mining and the flat memory model is that ARM performs at least some reasoning about the aliasing of memory locations outside the SMT solver, while the flat memory model offloads this work entirely to the solver. Consequently, the performance of the flat model is more subject to the peculiarities of the implementation of the underlying solver. We find that the flat model outperforms ARM in many small examples, but does not scale predictably. ARM scales more gracefully and can handle a larger fraction of verification tasks.

As a benchmark, we took 1128 verification tasks from our superoptimizer and performed verification twice, once with the alias relationship mining model, and once with the flat model. We find that the flat model timed out (after one hour) on 180 of them, but the ARM model timed out on only 24. However, for the verification problems where both models succeeded, the flat model had a much better average time of 24s per task compared to ARM with an average time of 554s per task.

In Sections 4.5.2 and 4.5.3 of this thesis we show how ARM enables superoptimization; without ARM, we cannot verify the correctness of the fastest optimized programs. ARM is also used to discharge the proof obligations described in Section 5.5 and is necessary to verify the correctness of some benchmarks without timeouts (see Section 5.5.6).

Chapter 4

Sound Loop Superoptimization for Google Native Client

4.1 Introduction

Software fault isolation (SFI) is a sandboxing technique to isolate untrusted code from a larger system [65, 12, 67, 56, 43]. Google Native Client (`NaCl`), a SFI system shipped with Google Chrome, safely allows untrusted extensions to be loaded into the web browser [67]. `NaCl` has been shown to be a robust real-world security technology; Google offers a $15,000 bug bounty for a sandbox escape [1]. SFI systems use a compiler to produce a specialized binary that obeys certain syntactic *rules*. To guarantee security, the SFI loader invokes a verifier to ensure the binary satisfies these rules. Specifically, the rules restrict the addresses of memory accesses and indirect jump targets at runtime.

However, performance is a "significant handicap" for SFI [56]. Currently, there is a performance penalty that all users of `NaCl` and other SFI implementations pay for the security guarantees. Building an optimizing compiler for SFI systems is difficult because of a pronounced phase ordering problem: Either a compiler can generate well-formed SFI code and then optimize it for performance, or it can generate optimized code and adjust it to obey the SFI rules. Both of these approaches lead to sub-optimal performance. Moreover, without verification, the optimizations have no formal guarantees of correctness [45, 39].

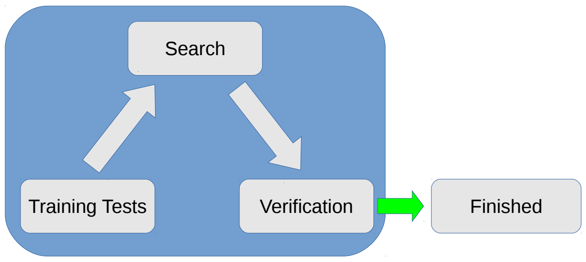
In contrast, search-based program optimization techniques start with a compiled *target* program, and attempt to search for a semantically equivalent *rewrite* with better performance characteristics. These tools, sometimes called *superoptimizers* or *stochastic superoptimizers*, make random modifications to the target code to generate candidate rewrites. The rewrites are evaluated with a *cost function* that estimates correctness, performance, and other properties. Correctness is generally estimated by running the code on test cases (either provided by the user or generated automatically). After an improved rewrite is found, a sound verification technique is used to verify correctness.

Superoptimization techniques address the phase-ordering problem by simultaneously considering a program's merit according to all desired criteria. Our hypothesis is that superoptimization can significantly improve SFI code generated by existing toolchains and offer a formal guarantee that the optimizations are correct. We demonstrate our hypothesis holds by optimizing frequently-used, and often performance critical, `libc` string functions that ship with `NaCl` (see Section 5.5). This chapter focuses on the technical problems we needed to solve to extend superoptimization techniques to this new domain.

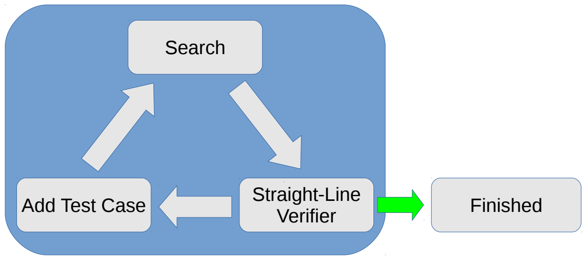
The key obstacle in applying existing superoptimization techniques to `NaCl` is simply the presence of loops in `NaCl` code. While prior work [59] extended superoptimization to small loop kernels of up to ten lines of x86-64 code, the loops that appear in real code are larger. We attempted to use the STOKE tool [53, 59, 55] for our initial `NaCl` superoptimizer. However, there are two general problems which caused the baseline STOKE to fail to optimize loops in `NaCl` code.

First, the search was guided exclusively by handwritten test cases. For small examples these tests suffice. However, more complex programs often have corner cases infrequently exercised by such tests. Past work on synthesizing straight-line code uses counterexamples from verification to guide the search (see Figure 4.1b). However, there is no general method for generating counterexamples from sound equivalence checking procedures for loops. The little existing work on search-based optimization of loops does not use automatically generated counterexamples when correctness proofs fail (see Figure 4.1a), greatly limiting its ability to generate correct rewrites.

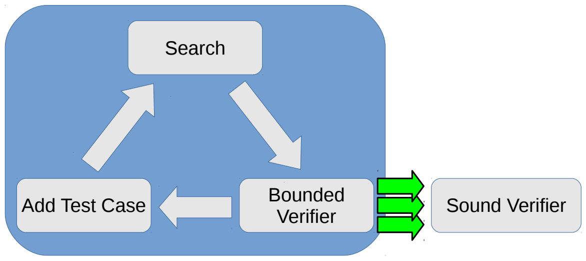
The second problem with our baseline implementation was an unexpected bias toward finding high-performance rewrites which were unlikely to be correct. The goal of the search is to find the fastest possible program that passes all of the test cases. Unsurprisingly, running the search for longer generally produces more performant results: over time, the search finds faster and faster programs. However, faster programs are also more likely to be overfitted to the test cases, and the search component has no mechanism to distinguish between correct and incorrect programs when all tests are passed. Retaining only the fastest such rewrite usually results in



(a) Search without automatic test case generation [59, 60].



(b) Search with automatic test case generation for straight-line code [53, 55].



(c) Search with two-stage verification and general automatic test case generation (this work)

Figure 4.1: Search architecture of different stochastic superoptimization tools.

failure during verification.

Our solutions to these two problems are simple, general, and result in dramatic improvements to our loop optimizer. We introduce a two-stage verification process with both a *bounded verifier* and a *sound verifier*, as shown in Figure 4.1c. A bounded verifier performs a partial proof of equivalence. For a user-supplied bound parameter k, it checks whether the target f and the rewrite g agree on *all* inputs for which every loop in each program executes for at most k iterations. If the check fails, then the bounded verifier produces a counter-example demonstrating the difference. Otherwise, the programs are equivalent up to the bound k, and the differences (if any) can only be demonstrated by running the bounded verifier with a higher k. In contrast, the sound verifier performs a sound proof of equivalence. Proving equivalence of two loops involves discovering and checking sufficiently strong loop invariants, which is fundamentally different from the approach taken by the bounded verifier. It is not obvious how to generate counterexamples from failed proofs of equivalence, nor is it even possible to do so in general.

We integrate the bounded verifier into the search loop to guide the search and identify the best rewrites. The stochastic search procedure generates successively improving rewrites that pass all of the current test cases and are estimated to perform better than the target. For each of these, we run the bounded verifier; if the bounded verifier accepts the rewrite, we add the rewrite to a store of *candidate rewrites* to formally verify later. Once the complete set of candidate rewrites is generated, we run the sound verifier on each of them, starting with the ones expected to perform best, until a provably correct rewrite is found. If, however, the bounded verifier fails, it generates a counterexample which is used as a new test case and is added to the running search. This approach helps guide the search from an incorrect rewrite to a correct one when the initial set of test cases is insufficient.

A limitation of bounded verification techniques is that their automatically generated test cases usually only run for a very small number of loop iterations. Consequently, we still require an initial set of longer-running tests to evaluate the performance of a rewrite. However, it is no longer necessary for user-supplied test cases to cover corner cases to guide the search to a semantically correct rewrite. The test cases we use in our experiments are generated randomly, rather than being hand-tuned as in previous work. The need for test cases to characterize performance is analogous to profile directed optimization [13, 3].

To evaluate our work, we implemented our new architecture as an extension to STOKE. We evaluate our implementation on a collection of 13 `libc` string functions shipped with the `NaCl` toolchain. We have chosen to focus on string and array benchmarks for three reasons. First, strings and arrays are ubiquitous in assembly code; any serious attempt at optimizing x86-64 assembly must handle them. Second, there are many applications where string and array functions are the chief performance bottleneck. Third, they present a real challenge, especially due to the possibility of arbitrary memory aliasing in the generated rewrites.

The formally verified binaries generated by STOKE improve performance on these production benchmarks by up to 97%, with a median and average of 25%. We also show that alias relationship mining (see Chapter 3) increases the number of verification tasks that can be completed. In summary, this chapter makes the following contributions:

- We demonstrate that stochastic superoptimization has a significant advantage over conventional compiler technology in optimizing SFI binaries. We achieve a median speedup of 25% across 13 `libc` binaries shipped with `NaCl` by Google. Our optimized binaries may be used as a drop-in replacement and are backed with a guarantee of formal equivalence to the original code.

- We introduce a new and robust architecture for stochastic program search for code containing loops. Our approach combines a bounded verifier with a sound verifier for proving loop equivalence. This is the first application of stochastic superoptimization to a real-world domain of loop functions.

- We detail enhancements to DDEC, the sound verification algorithm for proving loop equivalence introduced in [59]. DDEC is part of both our baseline and improved implementations. The enhancements make verification more robust in the presence of complex control flow.

The rest of the chapter is organized as follows. Section 4.2 offers an example of our techniques applied to one of our benchmarks. Section 4.3 details the implementation of the bounded and sound verification techniques. Section 4.4 discusses the extensions to STOKE required for generating `NaCl` code. In Section 4.5, we demonstrate our contributions empirically.

```
wchar* wcpcpy(wchar* edi, wchar* esi) {
  wchar* eax;
  do {
    wchar edx = *esi++;
    eax = edi;
    *edi++ = edx;
  } while (edx != 0);
  return eax;
}
```

Figure 4.2: C source for running example.

4.2 Motivating Example

Figure 4.2 gives C code for our running example, derived from the `wcpcpy` `libc` routine (string copy for wide character strings). In Figure 4.3 this function is compiled (the target code), and a rewrite is also shown. The rewrite is an example of an incorrect rewrite that may be produced by the STOKE superoptimizer. We will use this example to show the utility of bounded verification during search. Note that, even though the platform is 64-bit, `NaCl` treats all pointers as 32-bit. This example uses 32-bit wide characters.

The target and rewrite code in Figure 4.3 both obey the `NaCl` rules. In x86-64, an instruction is composed of an *opcode* and one or more *operands*. The opcode describes the functionality of an instruction, e.g., `mov`, `add`, etc. The suffix (e.g. `l` or `q`) denotes the width of the operands. An operand specifies what values to operate on. The operand can be a register (such as `%eax`), a memory operand (such as `(%r15,%rdi)`) or an immediate (a constant, like $4). Some details are:

- The register `%edi` points to the destination string and `%esi` points to the source string. The x86-64 ISA has 64-bit registers `%rdi`, `%rsi`, `%r15`, etc. The register `%edi` represents the lower 32 bits of `%rdi`. The `mov` instruction copies bits in the first argument to the second argument. Any instruction that writes to a 32-bit register also zeros the top 32 bits of the corresponding 64-bit register.

```
  # Target                            # Rewrite

.begin:                                movl %esi, %esi
  movl %esi, %esi                      movl (%r15,%rsi), %edx
  movl (%r15,%rsi), %edx               addl $4, %esi
  movl %edi, %eax                      nop (23)
  addl $4, %esi                       .begin:
  movl %edi, %edi                      movl %edi, %eax
  movl %edx, (%r15,%rdi)               movl %edi, %edi
  addl $4, %edi                        movl %edx, (%r15,%rdi)
  testl %edx, %edx                     shrl $1, %edx
  jne .begin                           je .exit
  retq                                 movl %esi, %esi
                                       movl (%r15,%rsi), %edx
                                       addl $4, %esi
                                       jmpq .begin
                                       nop (31)
                                      .exit:
                                       retq
```

Figure 4.3: A target and rewrite for `wcpcpy`. This benchmark performs a string copy of 4-byte wide characters.

For example, line 2 of the target leaves the lower 32 bits of `%rsi` unchanged and zeros the top 32 bits. This operation is important for the memory dereference at line 3 to be valid; `NaCl` requires memory operands to be of the form k_1`(%r15,`X`,`k_2`)`, where X is a 64-bit register whose top 32 bits are cleared by the previous instruction. This operand represents accessing memory at address $k_1 + \%\texttt{r15} + k_2 X$. When unspecified, $k_1 = 0$ and $k_2 = 1$.

- The `jne` on line 10 of the target redirects the control flow to line 1 if `%edx` is nonzero and to line 11 otherwise. A `jmp` redirects control flow unconditionally. `NaCl` has rules on instruction alignment. Hence, multi-byte no-ops are added. The notation `nop (X)` denotes a series of no-op instructions occupying `X` bytes.

- The `je` on line 10 of the rewrite jumps to the `.exit` label when `%edx` is 0 after the shift operation.

In the target there are two *basic blocks*, sequences of straight-line code delimited by labels and jumps: lines 1-10 (1_t); and line 11 (2_t). In the rewrite, there are four: line 1-4 (1_r); lines 5-10 (2_r); lines 11-14 (3_r); and the exit on lines 16-17 (4_r). A *path* through the program is a sequence of basic blocks that may be exercised by some input.

The rewrite code is almost correct, except that it computes the wrong jump condition. On line 9, it shifts the register `%edx` to the right by one, and branches if the result is zero. However, the target simply checks if `%edx` is zero; the rewrite is incorrect when the value of `%edx` is exactly one. In practice, if a wide string contains the character `0x00000001`, then the target performs the entire copy, but the rewrite stops early.

STOKE uses a cost function to guide it toward correct rewrites. To evaluate a rewrite, it runs it on inputs provided by the user. In previous work, if none of the user-provided inputs contains the character `0x00000001` (which is rarely used in practice) the search will not be guided away from this rewrite. This example is a realistic case where search, even guided by a robust collection of test cases, may still propose incorrect rewrites. We run the bounded verifier on rewrites that pass all test cases. When the bounded verification succeeds, the search continues; when it fails, we use the new counterexample as a test case which will guide the search away from the incorrect rewrite.

4.2.1 Bounded Verifier

The *bounded verifier* operates on this example as follows. For a given bound k, we enumerate the set of all possible paths through the target f and the rewrite g where no basic block repeats more than k times. For $k = 1$, there is only one path for each: $p_1 = 1_t 2_t$ and $q_1 = 1_r 2_r 4_r$. For $k = 2$, we have $p_2 = 1_t 1_t 2_t$ and $q_2 = 1_r 2_r 3_r 2_r 4_r$, in addition to p_1 and q_1.

For each target path p and rewrite path q, the bounded verifier checks if there is any input x for which the target executes path p, the rewrite executes path q, and the outputs of the two programs differ. In this case, the outputs are the return register `%rax` and the heap contents. If the two paths are *infeasible*, meaning there is no input x for which f executes p and g executes q, then the check is vacuously true. The bounded verifier builds a collection of *constraints* that express, as logical formulas, the relationships between the input x and the outputs of each program. Informally, we construct functions $f_p(x)$ and $f_q(x)$ representing the outputs of executing paths p and q on an input x. *Path conditions* $g_p(x)$ and $g_q(x)$ are predicates that express if paths p and q are taken on input x. Then, we use the Z3 SMT solver [21] to check if x exists such that $g_p(x) \wedge g_q(x) \wedge f_p(x) \neq f_q(x)$. If such an x exists, then we have generated a counterexample which can be used as a new test case for the search. Otherwise, f and g are equivalent for all inputs that execute paths p and q.

For $k = 1$ the bounded verification succeeds because the two programs are equivalent for the empty string. For $k = 2$, the bounded verifier checks equivalence for all runs executing the loops up to two times. When it compares p_2 to q_1, it produces a counterexample: for the input string with two 4-byte characters, the first one having value `0x00000001` and the second being a null character, the target and rewrite differ, as described earlier. This counterexample is then used as a new test case.

4.3 Implementation

This section describes the implementation of the bounded verifier, the alias relationship mining procedure, and the sound verifier.

4.3.1 Bounded Verifier

The *bounded verifier* takes a target f and a rewrite g and proves equivalence over a finite set of paths specified by a user-provided bound k. We generate sets of paths $Path_T$ and $Path_R$ through the target and the rewrite such that no basic block is repeated more than k times. For each $p \in Path_T, q \in Path_R$, we check that p and q are equivalent for all inputs that execute these two paths.

More formally, let x denote a *state*, a collection of sixteen 64-bit bitvectors (one for each general purpose register) and five boolean variables (one for each x86-64 flag). Memory is modeled via the alias relationship mining technique described in Chapter 3 or as an array using the flat memory model (Section 2.6). For each instruction s in our supported subset of the x86-64 instruction set, we have a function σ_s that describes the semantics of executing s on a state x [32]. Suppose path p executes instructions $s_1^f, \ldots, s_m^f$ through the target and path q executes instructions $s_1^g, \ldots, s_n^g$ through the rewrite. Let $x_0, \ldots, x_m$ and $y_0, \ldots, y_n$ denote the machine states as p and q are executed, and let x_f and y_f denote the final states. Then we generate the constraint $C \equiv C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$, where:

$C_1 \equiv x_0 = y_0$ constrains the input states to be equal. C_2 represents the execution of the target through p, i.e.,

$$C_2 \equiv x_f = x_m \wedge \bigwedge_{i=1}^{m} x_i = \sigma_{s_i^f}(x_{i-1})$$

C_3 represents the execution of the rewrite through q:

$$C_3 \equiv y_f = y_n \wedge \bigwedge_{i=1}^{n} y_i = \sigma_{s_i^g}(y_{i-1})$$

C_4 encodes path conditions; if s_i^f is a conditional jump `jf .L` (jump to `.L` if flag `f` is set) and the basic block following this instruction in p is labeled by `.L` then we generate a constraint asserting that `f` is set at x_{i-1}. Otherwise, we assert that `f` is unset at x_{i-1}. We do the same for the path q through g and conjoin all of these constraints. $C_5 \equiv x_f \neq y_f$ encodes that the output states of p and q differ on the output registers or the final heap state. C_6 is a conjunction of constraints that bound the address a of each memory dereference between $16 \leq a \leq 2^{64} - 16$. Counter-examples with very small and very large addresses are generally invalid.

We pass C to the Z3 SMT solver. A model for x_0 can be used as a test case demonstrating that the target and rewrite differ. If C is unsatisfiable, then the equivalence over p and q is proved and the bounded verifier analyzes the next pair of paths.

4.3.2 Sound Validation

Our sound verifier uses a strict definition of equivalence that is sensitive to termination, exceptions and memory side-effects. Let O be a set of output registers, and consider any program state x. We say that f is *equivalent* to g if, when we run f and g on x, exactly one of the following holds:

1. the target and rewrite both loop forever;

2. the target and rewrite both trigger a hardware exception;

3. the target and rewrite both execute to completion and terminate normally *and* the final states agree on output registers in O and all memory locations.

To perform sound verification, we extend previous work on data-driven equivalence checking (DDEC) [59], which uses test cases to guess a simulation relation between the target and the rewrite. An SMT solver is used to check the correctness of the simulation relation. If verified, the proof is complete.

The simulation relation is composed of *cutpoints* and *invariants*. A cutpoint is a pair of corresponding program points in the target and rewrite. Each cutpoint λ has an associated invariant ψ_λ that describes the relationship between states of the target and rewrite at λ. Our goal is to prove *inductiveness*; whenever we begin executing the target and rewrite from cutpoint λ on states x and y satisfying ψ_λ, the execution of the target and rewrite will both reach the same next cutpoint λ' in states x' and y' satisfying $\psi_{\lambda'}$.

We make the following improvements to the DDEC algorithm:

- When checking the inductiveness of the simulation relation using an SMT solver, DDEC enumerates all possible aliasing configurations, which is prohibitively expensive. We use alias relationship mining (see Chapter 3) to dramatically improve the efficiency of this step.

- DDEC can lose precision because it does not support disjunctive or inequality invariants, and its invariants never reason over memory. We add support for register-register inequalities, a restricted set of disjunctions, and invariants that assert a memory location is null. The additional precision is necessary to reason about branch conditions. DDEC had not previously been demonstrated on complete functions with multiple loops and branches.

- In some cases, invariants we learn from data are spurious. In [59] this would cause DDEC to fail. In this work, we have added fixedpoint iterations to eliminate spurious invariants, as in Houdini [27].

Choosing Cutpoints

The choice of cutpoints illustrates the correspondence between target and rewrite data that we use to learn an invariant. We use three types of cutpoints in the DDEC algorithm:

- a unique *entry cutpoint* at the entry to the program;

- a unique *exit cutpoint* at the exit of the program; and
- at least one *loop cutpoint* in every loop.

We model each program as having only one exit block, and transform every return statement as a jump to this block. To identify appropriate loop cutpoints, we perform a brute force enumeration of sets of pairs of program points. A set of cutpoints is *valid* if it satisfies four conditions: First, when the target and rewrite are executed on input x, they must reach the same cutpoints in the same order. Second, at each cutpoint, the heap-state of the target must agree with the heap-state of the rewrite. Third, there must be at least one cutpoint per loop. Finally, we only allow program points at the end of basic blocks to be cutpoints; this decision simplifies the implementation and makes the space of cutpoints to search smaller.

In some cases, DDEC fails with one set of cutpoints but succeeds with another. Therefore, if DDEC fails we run the algorithm again with a different cutpoint selection until all the possibilities are exhausted.

Learning Invariants

For each cutpoint λ, we guess a set of candidate invariants ψ_λ that relate the state of the target to the state of the rewrite when λ is reached. Given data from test cases (provided by the user or generated from counterexamples during search), we build a set S_λ of reachable state pairs (x_i, y_i) at λ. The invariant learning algorithm has two steps; first, we partition $S_\lambda = S_\lambda^1 \cup \cdots \cup S_\lambda^p$ based on control flow. Second, we learn the strongest set of predicates in our language of invariants that hold over each S_λ^j.

The partitioning is done based on control flow to derive useful disjunctive invariants. Suppose that the target and rewrite both have a conditional jump at λ. Let C_λ^t and C_λ^r denote predicates over states that express if the target (rewrite) takes the conditional jump. Then, we derive four partitions of S_λ corresponding to the different control flow outcomes for each state pair. Let $C_\lambda^1 = C_\lambda^t \wedge C_\lambda^r$, $C_\lambda^2 = \overline{C_\lambda^t} \wedge C_\lambda^r$, $C_\lambda^3 = C_\lambda^t \wedge \overline{C_\lambda^r}$ and $C_\lambda^4 = \overline{C_\lambda^t} \wedge \overline{C_\lambda^r}$. Define partitions $S_\lambda^j = \{(x_i, y_i) \in S_\lambda : C_\lambda^j(x_i, y_i)\}$.

$$Invariant := \sum_{i=1}^{n} A_i r_i = A_{n+1} \;\Big|\; r_1 < r_2 \;\Big|\; r_1 \leq r_2$$
$$\Big|\; r \neq 0 \;\Big|\; *mem = 0 \;\Big|\; r[64:32] = 0$$

Figure 4.4: Language of invariants used by DDEC algorithm. r is used to denote a 32 or 64-bit general purpose register and A denotes a bitvector constant. $*mem$ denotes a memory dereference. $r[64:32]$ denotes the top 32 bits of a 64-bit register.

If the target (or rewrite) does not have a conditional jump we merge the appropriate partitions.

For each set S_λ^j we learn the strongest set of invariants over pairs of states. These invariants are of the form $C_\lambda^j \Rightarrow \theta$, where the θ come from five classes of invariants as illustrated in Figure 4.4: (i) 64-bit affine bitvector equalities over registers; (ii) register-register inequalities; (iii) disequalities asserting a register is non-null; (iv) equalities asserting memory is null; (v) assertions that the top 32-bits of a 64-bit register are null.

Given S_λ^j, we find the strongest set of invariants in the language that hold over all state pairs. We use a dedicated algorithm for affine bitvector equalities, and a standard algorithm for the remaining invariant classes. The bitvector equality algorithm is as follows:

- Let L denote the set of live registers in the target and rewrite. Number these registers $0, \ldots, |L| - 1$.
- Build matrix M of size $|S| \times |L|$.
- Set M_{ij} to the value of register j in state pair (x_i, y_i).
- Apply Gaussian elimination adapted to bitvector arithmetic [23] to find a basis for all possible 64-bit affine equalities.

The other invariants can be learned from the test cases directly; for example, we check if, for some column, the top 32-bits of a register are zero in all the rows of the matrix; or, for each pair of registers r_1, r_2, if the relationship $r_1 < r_2$ always holds.

For each θ_i we have learned from S_λ^j, we add the invariant $C_\lambda^j \Rightarrow \theta_i$ to the candidate invariant set ψ_λ. Additionally, at every λ we add an invariant asserting the target and rewrite have identical heap states.

Inductiveness Check

We use the bounded verifier to perform the inductiveness checks soundly and efficiently. The candidate invariant ψ_λ is a set of predicates of the form $C_\lambda^j \Rightarrow \theta_i$. If some $C_\lambda^j \Rightarrow \theta_i$ is not inductive then it is removed from ψ_λ and the process is repeated until all remaining predicates are inductive. This process mimics the fixedpoint iterations performed by Houdini [27]. The fixedpoint iterations help discard any predicates that hold for the test cases but cannot be guaranteed to hold for all possible inputs. After reaching the fixedpoint, if the invariant established at the program exit cutpoint implies that the output states are equivalent, then we have successfully established the equivalence of the target and the rewrite.

4.4 STOKE for Google Native Client

The goal of STOKE's search algorithm is to find a rewrite that obeys the `NaCl` rules and produces the same outputs as the target on a given set of test cases. We extend the STOKE superoptimizer for this purpose.

At a high level, STOKE search is parametrized by the following: a search space of all possible rewrites, a cost function that uses test cases to identify preferable rewrites, and a set of transformations that can be applied to transform one rewrite in the search space to another. We run the search with a fixed number of iterations. In each iteration, we generate a new rewrite and evaluate a cost function. Depending on the cost, we either accept or reject the rewrite. For rewrites with lowest seen cost, we run the bounded verifier; if the bounded verifier says the target is equivalent to the rewrite, we add it to the output set of candidate rewrites.

To adapt STOKE for `NaCl`, we need to design an appropriate cost function to guide the search and add transformations relevant for `NaCl` to the existing transformations used by STOKE. These are described in the following subsections.

4.4.1 Transformations

Optimizing `NaCl` code requires more aggressive transformations compared to the ones described in previous works that use STOKE [53, 55, 59, 60]. In particular, previous work made no changes to the control flow. In this work, we relax this constraint and allow changes to jump instructions. We use opcode moves, local and global swaps, and instruction moves as described in [59]. Additionally, we include the following transformations:

1. *Operand moves* replace an operand of an instruction with a different one. This move also allows for jump instructions to change their targets. E.g., `jmpq` $.L1$ can be transformed to `jmpq` $.L42$.

2. *Rotate moves* move an instruction to a different place in the program.

3. *Opcode width moves* change an opcode and its operands to a similar instruction that operates on a different bitwidth. E.g., 32-bit `addl %eax, %ebx` can be transformed to 64-bit `addq %rax, %rbx`.

4. *Delete moves* remove an instruction entirely.

5. *Add no-op moves* insert an extra no-op into the program ($\star$).

6. *Replace no-op moves* replace an instruction with a string of no-ops whose binary representation has the same length as the original instruction ($\star$).

7. *Memory+Swap moves* replace a memory operand and simultaneously swap the preceding instruction with another one. ($\star$)

A ($\star$) denotes a transformation specific to `NaCl`. The *Memory+Swap* move is necessary because `NaCl` requires that the index of a memory operand is computed by the preceding instruction. Modifying either instruction alone is very likely to break this relationship. Therefore, there is a need for a single transformation that changes both simultaneously. The add and replace no-op moves help STOKE meet the alignment requirements of `NaCl` code. These specialized transforms required only 227 lines of additional C++ code.

4.4.2 Cost Function

A *cost function* produces a score for each rewrite, where lower values are better, and guides the search towards desirable rewrites. Our cost function is an aggregate of three different scores; one score measures compliance with the `NaCl` rules, another is for functional correctness and the last is for performance.

The `NaCl` score assigns a value of zero to well-formed `NaCl` code. To compute the penalty of alignment violations, we compute the minimum number of no-op bytes which must be added to or removed from the rewrite for it to follow the alignment constraints. To this end, we use a dynamic programing algorithm. For an

n-instruction rewrite, we build an $n \times 32$ matrix M where M_{ij} contains the minimum number of no-ops to be inserted or removed to align the ith instruction to j bytes beyond a 32-byte boundary while following all `NaCl` rules. The row M_{i+1} can be constructed from row M_i. The minimum value in row M_n is the alignment penalty. We also add fixed penalties (of value 100) for each ill-formed memory accesses, or the use of instructions unsupported by `NaCl`. We call the sum of these penalties the *nacl score*.

For functional correctness, we follow previous work [53, 55]: we run the rewrite on test cases and compare its outputs to those of the target on the same test cases. The *correctness score* is the Hamming distance between these outputs. Finally, the cost function includes a *performance score*. Previous work on STOKE uses a static approximation of performance. We compute a more accurate performance score by running the code in a sandbox on test cases and estimating the total runtime by summing precomputed latencies of each executed instruction. This score is more accurate because it is sensitive to the number of loop iterations. The precomputed latencies come from running an instruction in isolation on one core.

For each rewrite, the total cost is a weighted sum of correctness, performance, and nacl scores. For our benchmarks, we find the following function works well:

$$f = \begin{cases} \gamma * \text{correctness} + \text{nacl} + \text{performance} & \text{nacl} < \delta \\ \gamma * \text{correctness} + \eta * \text{nacl} + \text{performance} & \text{nacl} \geq \delta \end{cases}$$

We choose $\gamma = 10^6, \delta = 5, \eta = 25$. We do not believe that the particular constants are special; rather, a variety of different cost functions may work. We leave evaluating different designs of cost functions for future work. The implementation of this cost function required 434 lines of C++ code.

4.5 Evaluation

We use 13 `libc` string functions from the `newlib` library shipped with Google Native Client to evaluate our extensions to STOKE. We performed all experiments on machines with two Intel Xenon E5-2667v2 3.3GHz processors and 256GB of RAM.

We evaluate our work in three categories. First, we demonstrate that we can optimize these benchmarks and achieve formally verified `NaCl` code with a median and average speedup of 25%. Then, we compare the baseline STOKE implementation with our new system that uses the bounded verifier. Finally, we compare the performance of alias relationship mining (ARM) to the flat memory model.

4.5.1 Experimental Setup

Our goal is to improve the performance of each of the 13 `libc` string functions and prove correctness of the optimized code. For each benchmark, we perform two experiments: optimization and translation. In *optimization mode*, we initialize the rewrite with the code shipped with `NaCl` and run STOKE to improve its performance while maintaining compliance with the `NaCl` rules. In *translation mode*, the rewrite is initialized with code that does not comply with `NaCl` rules, and STOKE transforms it into well-formed `NaCl` code. For each benchmark, we assembled test cases from randomly generated strings.

The initial rewrite for the translation mode experiments is `gcc-4.9` code compiled for x86-64 with memory accesses systematically rewritten to follow `NaCl` rules on memory accesses; every access is written as a load-effective-address instruction to compute the sandboxed 32-bit pointer followed by a separate instruction that performs the dereference. The transformation helps STOKE find a rewrite faster, but it breaks correctness, degrades performance, and violates the alignment rules. However, starting here, STOKE is sometimes able to correctly translate such programs to correct and efficient `NaCl` code.

For each benchmark, we ran the search up to 15 times for 200,000 iterations each. We set a timeout of 6 hours on a single core per benchmark. This time is split between running search iterations and performing bounded verification to generate the candidate rewrites; summing across all benchmarks, about 2/3 of this time is spent in search, and 1/3 in bounded verification. All bounded verification is performed with a bound of $k = 1$. For each of the search runs, we run the sound DDEC verifier with a timeout of one hour on each candidate rewrite, in order of best expected performance, until we find one that verifies. Statistics on the benchmarks are in Table 4.1.

Performance Results

The performance results are shown in Figure 4.5 and Table 4.1. The improvements range from 0% (for `wcsrchr`) to 97% (for `wcslen`). The optimization and the translation results are incomparable. For some benchmarks, it is easier to optimize code that meets `NaCl` rules, and for others it is easier to translate already optimized code to valid `NaCl` code. However, the optimization experiment always succeeds (meaning we find a verified rewrite expected to be faster), while for several benchmarks, the translation experiment fails.

There were three common sources of optimizations. First, as seen in Section 4.2, many of the functions shipped with `NaCl` include instructions such as `movl %eax, %eax`; these do not perform any useful computation and their only purpose is to satisfy the `NaCl` rules on memory sandboxing (this instruction zeros the top 32 bits of the `%rax` register). STOKE is often able to use instructions, such as `addl $4, %eax` that meet the sandboxing constraints and perform necessary computations. Significant speedups are obtained when this change results in removal of an instruction inside a loop. This situation arises with the `wcslen` benchmark, where a speedup of nearly 2x is achieved by removing a single unnecessary instruction. Second, executing no-op instructions consumes processor cycles, and STOKE is sometimes able to move several no-op instructions outside of a loop to produce speedups. The original code has no-ops because `NaCl` enforces alignment rules, and moreover Google's `NaCl`

Benchmark	Target LOC	Best LOC	Best Speedup	Search Time (min)	DDEC Time (min)
wcpcpy	40	13	48%	37	38
wcslen	43	47	97%	78	89
wmemset	47	47	0%	29	45
wcsnlen	94	51	2%	61	83
wmemcmp	91	77	47%	360	302
wcschr	87	28	2%	61	5
strxfrm	99	38	0%	81	414
wcscmp	108	29	47%	38	586
wmemchr	132	75	2%	67	30
wcscpy	35	40	25%	276	252
wcscat	89	90	26%	360	46
strcpy	70	63	30%	360	415
wcsrchr	178	178	0%	30	15

Table 4.1: Performance results for verified benchmarks. LOC shows how many lines of assembly codes in the target program. “Best LOC” and “Best Speedup” show the number of lines of code and the speedup for the best rewrite found. The search time includes both search and bounded verifier queries for the optimization mode task. The DDEC time shows the total time required to complete all sound verification tasks in optimization mode.

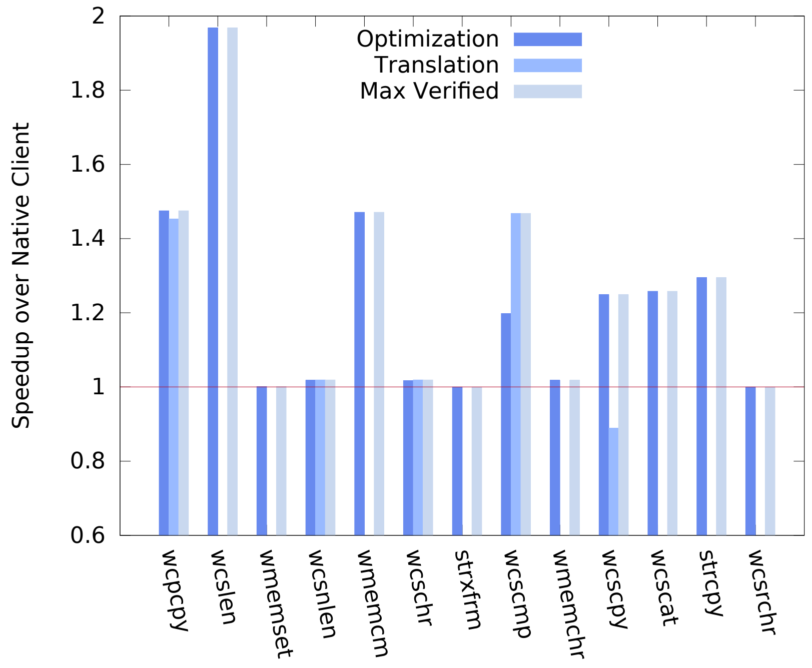


Figure 4.5: Speedups by benchmark. For each benchmark, the speedup over the original `NaCl` library is shown. The bars correspond to the optimization experiment, the translation experiment, and the best rewrite we verified. The 'optimization mode' much more reliably produces a verifiable result, but 'translation mode' sometimes offers significant improvements.

compiler is overly conservative: it aligns every jump to a 32-byte boundary instead of only indirect jumps. Table 4.1 shows that even though code size was not measured in the cost function, STOKE reduced the aggregate code size by about 30%. Third, although `gcc` generally does well, STOKE sometimes improves register allocation and instruction selection.

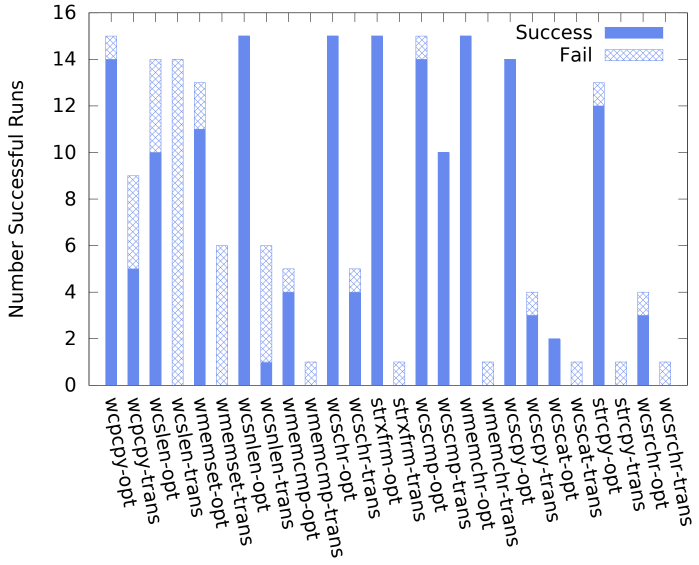
In the case of `wcscmp`, with a translation mode speedup of 47%, both removing no-ops and improving the use of sandboxing instructions made the code much smaller – 29 lines down from 108. In the target, the loop contained 40 instructions (mostly no-ops), but the translation mode rewrite loop contains only 10. This reduction has a significant impact at the architectural level; we believe this change allowed the processor's loop stream detector to optimize code execution.

Verification Results

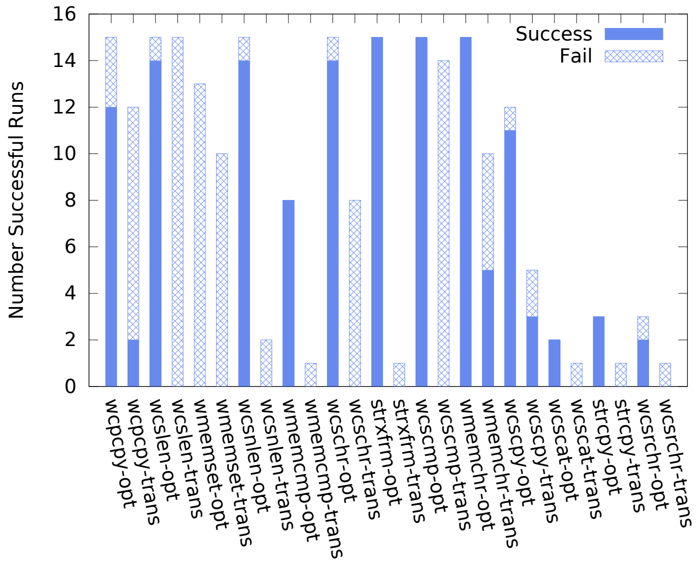
In optimization mode, STOKE always finds and verifies a rewrite for every benchmark. However, the translation mode benchmarks infrequently produced a verified rewrite, for two reasons. First, the translation mode search starts with a program that does not obey `NaCl` rules, and the search has to fix this discrepancy before it can produce any rewrite. As a result, it may take much longer for the translation mode experiment to find a first rewrite.

Second, the start program for translation mode is semantically different from the target. We used `gcc-4.9` with full 64-bit pointers, while the `NaCl` compiler uses 32-bit pointers. As a result, bitwidths for different instructions differed between the target and the rewrite. In many cases, the search would produce rewrites that were almost correct; they would be equivalent for all input strings of up to 2GB in size, but would fail for larger strings. Often, an unsigned length was treated as a signed value, and vice-versa. The bounded verifier could not guide the search in these cases because it could only produce small test cases. However, the DDEC verifier rejects such "almost correct" rewrites. Yet sometimes, the code generated by `gcc-4.9` is closer to a fast rewrite than the code generated than the `NaCl` compiler, and we obtain strong performance results.

Figure 4.6 shows end-to-end results for search and verification, including the number of candidates from search, and the number of verification successes and failures. The verification failures were for two reasons. In only one case, the verification timed out on a correct rewrite; this instance is for `wcpcpy` in translation mode. For the other 180 failures, the candidate rewrite was indeed not equivalent; this problem was particularly frequent for translation mode benchmarks as described in the previous paragraph. In 20 of these 180 cases, the rewrite was both incorrect and the solver timed out. It is to be expected that incorrect rewrites are more likely to cause a timeout because the modified DDEC algorithm will continue to search for more cutpoints until they have all been exhausted or time expires. Never did the verification fail for a correct rewrite, suggesting our choice of cutpoints and loop invariants were



(a) Alias Relationship Mining



(b) Flat Memory Model

Figure 4.6: End-to-end search and verification success rates. For each benchmark, the total height of the bar shows the number of candidate rewrites found during search. We apply the sound verifier to each candidate. The "success" measure shows how many verification tasks succeeded; the "fail" measure shows how many failed. In optimization mode, we always find verified results with the ARM model, but not with the flat memory model. Note that the memory model affects the search in addition to the verification, because it impacts which counterexamples are generated.

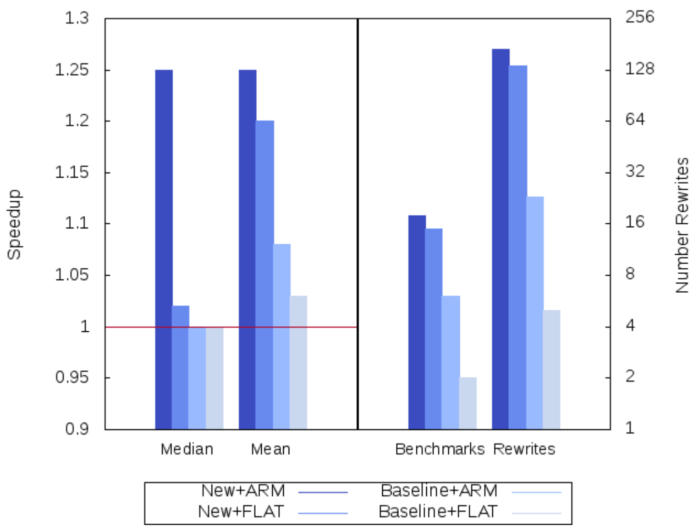


Figure 4.7: Comparison of implementations. For each of four implementations, the median and mean best speedups are plotted across the 13 benchmarks in translation and optimization modes. The number of benchmarks where an improved result was found (maximum is 26 considering optimization and translation mode), and the number of candidate rewrites generated by the search is also shown; these last two figures are plotted against a log scale. Using the bounded verifier to generate a stream of candidate rewrites substantially improves the quality of the rewrites; using ARM also improves the number and quality of verified rewrites.

sufficient.

One observes that the 180 cases where sound verification failed due to an incorrect candidate rewrite contradicts the often-assumed "small scope hypothesis" [34, 47, 2]. This hypothesis says that if the program is correct for small inputs, then it is likely correct for larger inputs too. This hypothesis fails for our domain of simple `libc` string functions. Often the bugs are very subtle and only appear for large inputs; without the sound validator, we are unlikely to find them.

4.5.2 Comparison to Baseline Implementation

We re-ran the experiment using our baseline implementation. The baseline implementation does not use the bounded verifier at all. Instead, the search runs for a fixed number of iterations and returns the best rewrite that passes all the test cases. Then, the sound verifier is used to check for correctness. We ran the experiment once with the ARM memory model and once with the flat model.

With the baseline implementation and ARM memory model, we only obtain results for four benchmarks in optimization mode, namely `wcpcpy`, `wcscmp`, `wmemchr`, and `strcpy`, with corresponding speedups of 48%, 47%, 2%, and 0%. For the other 9 benchmarks, the search results could not be verified because they were incorrect. Without the bounded validator, we only obtain an average speedup of 7%. Across all 13 benchmarks, the bounded verifier implementation with ARM generates 168 verified rewrites of varying performance, but the baseline implementation with ARM only generates 23.

The baseline does poorly for two reasons. First, there is no bounded verifier to help guide the search. Second, the search only returns the rewrite with the best performance estimate, and discards the potentially valuable intermediate results that are more likely to be correct.

Figure 4.7 shows aggregate statistics for four different implementations: the baseline and bounded verifier implementations, each run with the ARM and flat memory models. The median and mean speedups are shown, along with the total number of benchmarks with a verified result (counting both optimization and translation mode), and the total number of verified rewrites found. Using a bounded validator and ARM yields an 8.3x improvement on mean speedup over the baseline implementation with a flat memory model.

4.5.3 Memory Model Performance

Figure 4.6b shows success rates for search and verification when the flat memory model is used instead of ARM. The `wmemset` benchmark is of particular interest. Out of the 13 successful runs, the search generated 66 candidate rewrites. Of these, 49 invocations of DDEC timed out after 1 hour. The other 17 candidates failed to verify because of errors in the rewrites. On the same set of 49 verification tasks that timed out, running DDEC with ARM succeeds, and each one finishes in under 6 minutes. Similarly, ARM succeeds on five of the translation mode benchmarks, but the flat model only succeeds on three of these. In particular, the flat model times out on the `wcscmp` benchmark while ARM succeeds; this benchmark also is the one with the greatest performance improvement in translation mode.

Chapter 5

Semantic Program Alignment

5.1 Introduction

Program equivalence checking is commonly performed in two stages: the first stage is to construct a *product program* for the two programs by *aligning* them, and the second is proving a safety property, or *invariant*, of the resulting program [9, 68]. There is a trade-off between the effort put into each stage. For example, consider the functions f and g in Figures 5.1a and 5.1b, where each function iterates over some loop-free basic block. A simple product program for f and g is shown in Figure 5.1c, where one program is run after the other; one may check the invariant that the outputs of f and g are equal. However, this alignment provides no help in checking equivalence, which requires completely summarizing the loops of f and g.

In some cases, a better alignment is to pair iterations of the loops of f and g, as pictured in Figure 5.1d. This alignment sometimes facilitates an easier, inductive proof of equivalence in which corresponding loop-free code fragments are shown to preserve an invariant Inv for each loop iteration [59]. However, such syntactically constructed alignments only work in simple cases where the loops of f and g execute for the same number of iterations. This is not the case for several common loop optimizations that alter syntactic structure, such as vectorization, loop unrolling, and loop peeling. What is needed is a semantically guided alignment that relates the two programs in a way that is designed to make the final proof of equivalence as simple as possible.

We introduce a novel and robust technique for constructing product programs driven by semantics, rather than syntax, that extends equivalence checking to real-world benchmarks that are beyond the reach of prior work. Given two functions f and g along with test cases provided by the user, we build a *trace alignment*, which is a pairing of states in execution traces of f and g for each user-provided test case. Constructing the trace alignment is guided by the selection of a weak invariant, called an *alignment predicate*, that identifies pairs of machine states that should be aligned. Only once we have identified a trace alignment based on semantic properties of the programs do we lift the alignment back to the program syntax,

```
f() {
  while(*)
    A;
  return a;
}
```

(a) Function f

```
g() {
  while(*)
    B;
  return b;
}
```

(b) Function g

```
X() {
  while(*)
    A;
  while(*)
    B;
  assert(a == b);
}
```

(c) Naive Composition

```
Y() {
  while(*)  {
    assert(Inv);
    A;
    B;      }
  assert(a == b);
}
```

(d) Syntactic Composition

Figure 5.1: Two functions and two product programs. A and B are basic blocks and share no variables.

construct a product program, and learn invariants that we attempt to prove. This approach to constructing the product program, wherein we first solve the problem of semantically aligning the traces, is the novel contribution that allows us to verify equivalence where techniques described in prior work are inapplicable.

Our goal is to perform black-box verification of optimizations performed by compilers, superoptimizers, or by hand, without any foreknowledge of the transformations applied or toolchains used. Therefore, we evaluate our technique directly on `x86-64` assembly. Given two functions, our technique utilizes a set of user-provided test cases to guess a set of candidate alignment predicates. For each alignment predicate, we infer the trace alignment and attempt to construct a *program alignment automaton* (PAA) that specifies a product program. We again use the test cases to learn the invariants of the PAA. Finally, we use an SMT solver to check proof obligations that establish the equivalence of the functions.

We demonstrate the ability to verify several types of loop optimizations, including loop unrolling, loop peeling, vectorization, software pipelining, strength reduction, loop-invariant code motion, register allocation and loop inversion, among others. We evaluate our technique on 56 realistic loop benchmarks where compilers (`gcc`-4.9.2 and `clang`-3.4) automatically perform a number of these optimizations, at least including vectorization. We further apply our technique to verify the correctness of the hand-vectorized C implementation of the `strlen` function that ships with GNU C Library (`libc`, version $\geq$ 2.10.1), and also show that our method can verify benchmarks used to evaluate other state-of-the-art equivalence checkers.

In this chapter we offer the following contributions:

- A novel and robust approach for semantics-driven construction of product programs using alignment predicates and trace alignments.
- A set of 56 realistic `x86-64` benchmarks for evaluating equivalence checking techniques on optimizations that alter control flow, such as loop unrolling, loop peeling and vectorization.

- The first fully automatic black-box algorithm for proving the correctness of vectorization optimizations as performed by modern compilers on `x86-64`.
- A demonstration of a useful, real-world application of our equivalence checking technology to verify the correctness of a handwritten vectorized implementation of the `strlen` function shipped in `libc`.

The rest of this chapter is structured as follows. First, we introduce our running example (Section 5.2) before presenting the formalisms used in our work (Section 5.3). Then follows our equivalence checking procedure (Section 5.4) and evaluation (Section 5.5).

5.2 Example

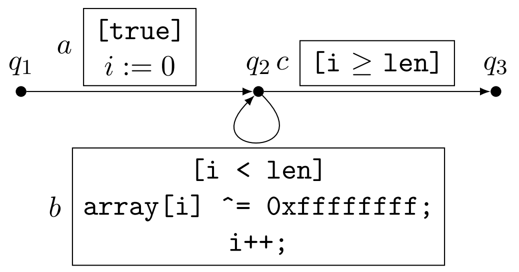
Consider the pair of C programs in Figure 5.2. Each function is represented as a control flow graph (CFG); the nodes are program points, and the edges are basic blocks along with a guard predicate. These functions take as input two parameters: `array`, which points to an array of 32-bit integers, and `len`, which specifies the length of the array. Both functions flip the bits of each array element. Function f (Figure 5.2a) iterates over each element in the array with a counter variable `i`, while g (Figure 5.2b) illustrates a simple way to vectorize this code using a 64-bit operation. In g, the loop body c' (lines 8-10) flips the bits of two array elements. Before the loop, there are two possibilities. If `len` is odd, block a' (lines 3-5) executes and flips the bits of the first array element. Otherwise, block b' executes and leaves the array untouched.

This example is representative of a number of challenges that naturally arise in the presence of loop optimizations. For example, if the loop in g iterates n times, then the loop in f iterates for either $2n$ or $2n + 1$ iterations, depending on the parity of `len`. Previous equivalence checking techniques handle situations where the relationship between the number of iterations of f and g is static (e.g. if g iterates n iterations then f iterates $2n$ iterations for all inputs). As a result, prior automated equivalence checking approaches [17, 59, 8, 48, 25, 26] fail on this example.

Our technique requires as input a set of test cases $\tau_1, \ldots, \tau_n$. The test cases may, for example, be generated randomly or by bounded model checking. We execute f and g on each test case to obtain traces. Figure 5.3 shows traces for each program with `array` initialized to address `0x100000` and `len=5`.

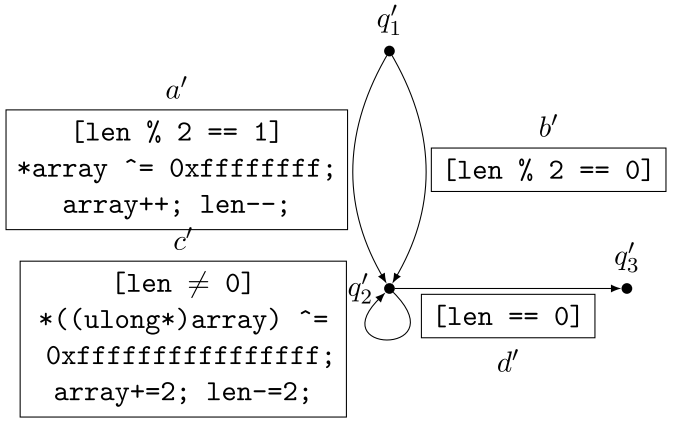
We begin by guessing an *alignment predicate*, ξ, over pairs of machine states from f and g that will help us align traces of f and g when run on the same input. For this example, consider the alignment predicate $\xi = \{$`array + 4i` $=$ `array'`$\}$. Suppose ρ and ρ' are traces of f and g for a particular test case. We consider a machine state σ from ρ and σ' from ρ'. If the predicate $\xi(\sigma, \sigma')$ holds, we say the two traces are *aligned by* ξ at that pair of states. Additionally, we consider ρ, ρ' to be aligned at the

```
void f(int* array, uint len) {
  for(uint i = 0; i < len; i++)
    array[i] ^= 0xffffffff;
}
```



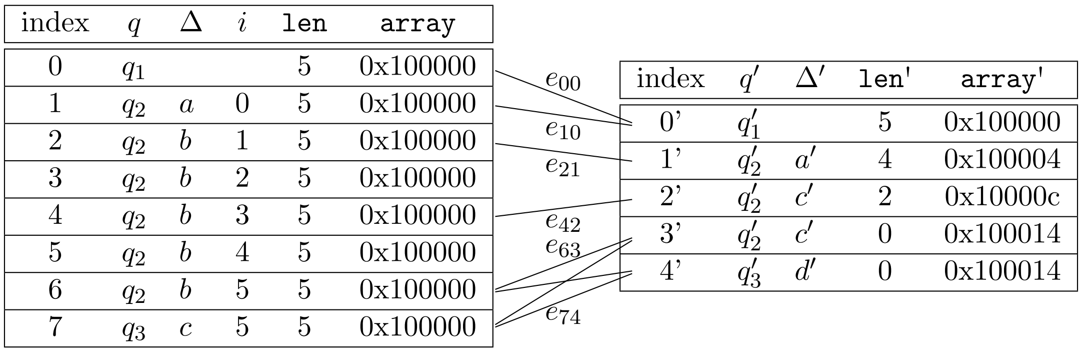
(a) C source and CFG for the unoptimized program, f.

```
void g(int* array, uint len) {
  if(len % 2 == 1) {
    *array ^= 0xffffffff;
    array++;
    len--;
  }
  while(len) {
    *((long*)array) ^= 0xffffffffffffffff;
    array += 2;
    len -= 2;
  }
}
```



(b) C source and CFG for the vectorized program, g.

Figure 5.2: Functions f and g used in the example.



index	q	Δ	i	`len`	`array`
0	q_1			5	0x100000
1	q_2	a	0	5	0x100000
2	q_2	b	1	5	0x100000
3	q_2	b	2	5	0x100000
4	q_2	b	3	5	0x100000
5	q_2	b	4	5	0x100000
6	q_2	b	5	5	0x100000
7	q_3	c	5	5	0x100000

index	q'	Δ'	`len'`	`array'`
0'	q'_1		5	0x100000
1'	q'_2	a'	4	0x100004
2'	q'_2	c'	2	0x10000c
3'	q'_2	c'	0	0x100014
4'	q'_3	d'	0	0x100014

Figure 5.3: Execution traces of f and g for a particular input. Column q shows the program point and column Δ shows the last basic block executed. The edges indicate pairs of states where the alignment predicate $\texttt{array} + 4\texttt{i} = \texttt{array'}$ holds.

beginning and at the end, even if ξ does not hold. In Figure 5.3 we have drawn edges between every pair of states in the two traces that are aligned by ξ. Observe that the alignment may pair states in a many-to-many correspondence, and that edges may cross. A *trace alignment* by ξ is obtained by performing this procedure for a set of test cases.

The trace alignment gives pairs of *corresponding paths* that relate the behavior of f with g as follows. Consider any two edges e_i, e_j in Figure 5.3 that do not cross each other and have no edges in between them (e.g. e_{10} is "between" e_{00} and e_{21}, while e_{64} and e_{73} cross each other). Each of e_i, e_j is associated with a machine state in the execution trace of f. These two machine states delimit some series of basic blocks, called a *path*, in f. Similarly, e_i and e_j delimit a *corresponding path* in g. For example, consider edges e_{21} and e_{42}. Between index 2 and index 4 of the trace of f, the path *bb* is executed, while between index 1' and 2' of the trace of g, block c' is executed. Thus, *bb* and c' are corresponding paths. Table 5.1 lists some of these pairs of edges and the corresponding paths.

We use the corresponding paths to build a *program alignment automaton* (PAA) that (we hope) overapproximates the behaviors of both programs. The PAA has one node for each pair of program points. For each pair of corresponding paths in

Edges	States	P	Q
$e_{00} \to e_{10}$	$q_1q_1' \to q_2q_1'$	a	ϵ
$e_{10} \to e_{21}$	$q_2q_1' \to q_2q_2'$	b	a'
$e_{21} \to e_{42}$	$q_2q_2' \to q_2q_2'$	bb	c'
$e_{42} \to e_{63}$	$q_2q_2' \to q_2q_2'$	bb	c'
$e_{63} \to e_{64}$	$q_2q_2' \to q_2q_3'$	ϵ	d'
$e_{63} \to e_{73}$	$q_2q_2' \to q_3q_2'$	c	ϵ
$e_{64} \to e_{74}$	$q_2q_3' \to q_3q_3'$	c	ϵ
$e_{73} \to e_{74}$	$q_3q_2' \to q_3q_3'$	ϵ	d'

Table 5.1: Pairs of edges and corresponding paths.

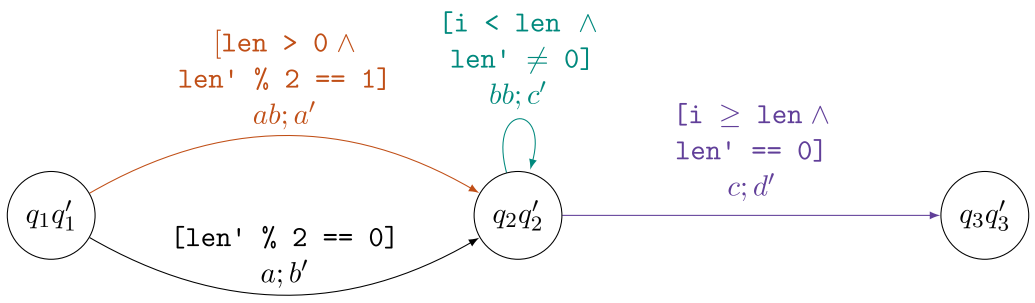


Figure 5.4: Simplified program alignment automaton for the example. The colors show the correspondence between the transitions and the pairs of corresponding paths in Table 5.1.

each pair of traces, we add a transition to the PAA labeled by these two paths. We perform a greedy simplification procedure to remove redundant nodes and edges (see Section 5.4.2). For the example, we remove the nodes q_2q_1', q_2q_3' and q_3q_2', and concatenate their incoming and outgoing transitions (so transitions $q_1q_1' \to q_2q_1'$ and transitions $q_2q_1' \to q_2q_2'$ are replaced by transitions $q_1q_1' \to q_2q_2'$). Note that different alignment predicates will define very different PAAs.

Figure 5.4 shows the simplified PAA for the example, which characterizes all the program behaviors. The three nodes are the natural outcome of the construction after simplification. This is in contrast to prior work such as [14, 59, 17] where corresponding points in the two programs, sometimes called *cutpoints*, must be chosen based on less information; usually cutpoints are chosen syntactically. Our construction guarantees that ξ holds in all nodes of the PAA (except possibly the entry and exit nodes) for all the traces generated by the test cases.

In Figure 5.4, there are two transitions between q_1q_1' and q_2q_2'; one is for inputs where `len` is odd, and a' is executed in g. The other is for inputs where `len` is even, and b' is executed in g instead (this transition comes from corresponding paths of aligned traces where the starting value for `len` is even). The paths labeling the transitions show that b is executed in f an extra time if a' is executed. Each path P has a path condition ψ_P, which is the conjunction of the predicates on its basic blocks. Each transition λ labeled by paths P, Q has a path condition given by $\psi_\lambda = \psi_P \wedge \psi_Q$, as shown in Figure 5.4.

Our next goal is to learn an invariant ϕ_s at every node s in the PAA and then prove that the PAA soundly overapproximates both programs. At the start node, we fix the invariant to assert equality of the input registers and the initial heaps. At the exit, we assert equality of the output registers and the final heaps. The other invariants are learned using the execution traces from the test cases provided by the user (Section 5.4.4). A subset of the learned invariants for the example is shown in Figure 5.5.

The choice of alignment predicate is crucial to finding invariants. For example, suppose that our alignment predicate depicted in Figure 5.3 also paired state 3 of f's trace with state 1' of g's trace. After simplification, there is a transition $q_1q_1' \rightarrow q_2q_2'$ labeled by $abb; a'$. Consequently, *none* of the invariants for q_2q_2' depicted in Figure 5.5 would hold. Instead, to prove equivalence one would need a set of disjunctive invariants to reason about two cases: either f is one iteration ahead of g, or it is not (depending on whether the transition labeled $abb; a'$ is taken). A similar problem arises if one labels a transition $a; a'$ instead of $ab; a'$, as is the case in works such as [59, 8], where loop iterations are assumed to be in one-to-one correspondence.

To prove the equivalence of the two programs, there are two primary types of proof obligations we must check (see Section 5.3.1). First, we check that the invariants

$$\begin{aligned}
\phi_{q_1q_1'} &:= \mathit{array} = \mathit{array}' \wedge \mathit{len} = \mathit{len}' \wedge \omega = \omega' \\
\phi_{q_2q_2'} &:= \mathit{array}' - 4i = \mathit{array} \wedge \mathit{len} - i = \mathit{len}' \wedge \\
&\quad\; i \leq \mathit{len} \wedge \omega = \omega' \\
\phi_{q_3q_3'} &:= \omega = \omega'
\end{aligned}$$

Figure 5.5: Invariants needed for the example. ω and ω' denote heap states of f and g. The alignment also allows us to show that $\mathit{len}' \equiv 0 \pmod 2$ at q_2q_2', although this fact is unneeded.

hold. For each transition $s \to t$, with paths P and Q, we verify the following: if a pair of machine states satisfies ϕ_s, then if paths P and Q are executed, the execution terminates without error in states satisfying ϕ_t.

Second, we must ensure that the PAA has the necessary transitions to overapproximate all program behaviors. Each node s corresponds to a pair of program points (q_i, q_j'). We want to ensure that every pair of feasible execution paths starting at q_i and q_j' is represented in the automaton. Consider the node q_2q_2'. From node q_2 there are two kinds of executions: (α), those for which $i \geq \mathit{len}$ and execution halts; and (β), those for which $i < \mathit{len}$ and execution continues. From q_2' there are similarly two kinds of executions: (γ), those for which $\mathit{len}' = 0$ and execution halts; and (δ), those for which $\mathit{len}' \neq 0$ and execution continues. Thus there are four pairs of possible behaviors: $\alpha\gamma, \alpha\delta, \beta\gamma$ and $\beta\delta$. Of these, $\alpha\gamma$ and $\beta\delta$ are already represented in the PAA via the self-loop at q_2q_2' and the transition $q_2q_2' \to q_3q_3'$. For the other two, we need to show they are infeasible. Now, $\alpha\delta$ only executes if $i \geq \mathit{len}$ and $\mathit{len}' \neq 0$, however $i \geq \mathit{len} \wedge \mathit{len}' \neq 0 \wedge \phi_{q_2q_2'}$ is unsatisfiable. Similarly $i < \mathit{len} \wedge \mathit{len}' = 0 \wedge \phi_{q_2q_2'}$ is also unsatisfiable, so $\beta\gamma$ is infeasible.

Verifying these proof obligations is sufficient to conclude that these two programs are equivalent for all inputs.

5.3 Formalization

We say that two `x86-64` functions, f and g, are *equivalent* if, when run starting in identical machine states (registers, stack, heap), one of the following holds:

1. both terminate normally, with identical heap-state and identical output registers; or
2. each program either encounters a run-time error or loops forever.

We use σ to denote a machine state, including the program counter, and all register, stack and heap values. We use ω to denote just the heap. When we use x to denote a state element of f, we use x' to denote the corresponding state element of g. A trace, ρ, is a sequence of machine states.

We use relational Hoare triples [10, 33] to express proof obligations. Let ϕ_1, ϕ_2 be predicates on pairs of machine states from f and g, and let P (resp. Q) be a path through f (resp. g). Then $\{\phi_1\}\ P\ ;\ Q\ \{\phi_2\}$ denotes the statement: if $\phi_1(\sigma, \sigma')$ holds for states σ, σ' of f, g and paths P, Q are executed (implying the path conditions hold), then execution terminates normally in states σ'', σ''' where $\phi_2(\sigma'', \sigma''')$ holds.

A PAA is an automaton where each node s is labeled with a pair of program points, one from f and one from g, along with an invariant ϕ_s. We assume that f and g each have unique entry and exit program points. The start node of the PAA corresponds to the pair of program entries, and the unique final node corresponds to the pair of exit points. Each transition is labeled by a finite path in each program: a transition $(u, u') \rightarrow (v, v')$ must be labeled with a path P in f from u to v, and a path Q in g from u' to v', where either P or Q must be nonempty. The PAA can be thought of as a control flow graph for the product program, although we do not explicitly build the product program in our work.

5.3.1 Proof Obligations for Program Alignment Automata

To use a PAA to check program equivalence, the following properties must be verified:

1. For each transition $s \to t$ labeled with paths P, Q, it holds that $\{\phi_s\}\ P\ ;\ Q\ \{\phi_t\}$.
2. For each node $s = (u, u')$, all pairs of program paths through f and g starting from u and u' not included in the PAA must be infeasible. That is, if P and Q are paths through f and g starting at u and u', and there is no transition $s \to t$ labeled by P^*, Q^* where P^* is a prefix of P and Q^* is a prefix of Q, then $\{\phi_s\}\ P\ ;\ Q\ \{\text{false}\}$.
3. The PAA has no cycles of transitions where all the paths through f or g are empty.
4. The invariant of the final node implies that the heap states and output registers are equal.

The alignment predicate, and the trace alignment derived therefrom, play the critical role of selecting the right transitions for the PAA so that we can prove the invariants at each node. The following lemma illustrates the key inductive argument for a proof of equivalence, and the corollary establishes soundness.

Lemma 1. *Let A be a program alignment automaton where the above proof obligations have been checked. Suppose f and g are executed from states σ, σ' at the program points (u, u'), and there is a node $s = (u, u')$ of A where $\phi_s(\sigma, \sigma')$ holds. Then if f executes to completion without exceptions within m steps (each step is an execution of a basic block), g also executes to completion without exceptions, and their final states satisfy the invariant of the final node of A.*

Proof. By strong induction on m. When $m = 0$, the premises imply that f and g have already executed to completion and the conclusion holds. Suppose the lemma holds for $0 \leq i < m$. Assume f and g are executed from (u, u') and that f terminates within m steps. By proof obligation 2, some prefix of the execution traces of f and g must match the paths P, Q of some transition $\lambda : s \to t$ in A. Removing these prefixes from the execution traces, we now have a new pair of traces where f and

g execute from t with states σ'', σ''' that satisfy ϕ_t (by proof obligation 1). In the case where P is non-empty, f still executes to completion, but now within $j < m$ steps. By the inductive hypothesis, we can conclude the lemma holds. In the case where $P = \epsilon$, we repeat the step of identifying a matching transition and removing the trace k times, where k is the length of the longest series of transitions from s where the paths for f are empty; by proof obligation 3, k must be defined. Then proceed as before. □

Corollary 2 (Soundness)**.** *If there exists a program alignment automaton, A, for f, g where the proof obligations hold, then f and g are equivalent.*

Proof. Suppose we run f, g on an input. By Lemma 1 if f terminates without error then g does also; by swapping f and g the converse also holds. The lemma also implies the final invariant of A holds, and by the fourth proof obligation f and g are equivalent. □

```
function VERIFY($f, g, data$)
    $(d_{train}, d_{test}) \leftarrow$ Partition($data$)
    $AP \leftarrow$ GuessAlignmentPredicates($f, g, d_{train}$)
    for all $\xi \in AP$ do
        $TA \leftarrow$ BuildTraceAlignment($\xi, d_{train}$)
        $A \leftarrow$ BuildPAA($TA$)
        if TestPAA($A, d_{test}$) then
            $A \leftarrow$ LearnInvariants($A, d_{test} \cup d_{train}$)
            if CheckProofObligations($A$) then
                return $equivalent$
            end if
        end if
    end for
    return $unknown$
end function
```

Figure 5.6: The equivalence checking algorithm.

5.4 Equivalence Checking Procedure

Figure 5.6 gives pseudocode for our algorithm. The user supplies two functions, f and g, along with a set of test cases, *data*. The test cases are partitioned into two sets, a training set and a test set. We invoke `GuessAlignmentPredicates` with the training data to build a set of candidate alignment predicates (Section 5.4.6). For each alignment predicate ξ we call `BuildTraceAlignment` to construct a trace alignment, TA, over the training data (Section 5.4.1) and then use TA to construct the PAA (Section 5.4.2). To ensure that the PAA is general and not overfitted to the training data, we use the test data to check the *viability* of the PAA via `TestPAA` (Section 5.4.3). Finally, we learn the invariants for the PAA (Section 5.4.4) and check the proof obligations (Section 5.4.5).

5.4.1 Construction of the Trace Alignment

Given an alignment predicate ξ we construct a trace alignment. The trace alignment TA is a set of pairs (ρ, ρ') where ρ and ρ' are prefixes of the traces of f and g for some test case. We initialize TA to the empty set. For each training test case τ we execute f and g to obtain traces $\rho_\tau = \sigma_1\sigma_2\cdots\sigma_n$ and $\rho'_\tau = \sigma'_1\sigma'_2\cdots\sigma'_m$. For each σ_i, σ'_j we check $\xi(\sigma_i, \sigma'_j)$; when satisfied, we add the pair $(\sigma_1\sigma_2\cdots\sigma_i, \sigma'_1\sigma'_2\cdots\sigma'_j)$ to TA. Figure 5.3 shows prefixes of traces that are aligned by ξ in the example.

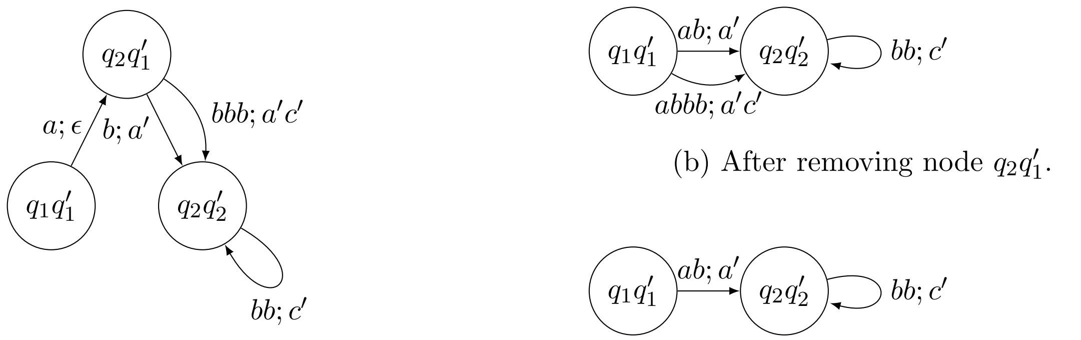
5.4.2 Construction of the Program Alignment Automaton

We initialize the PAA with a node for every pair of program points in the two programs. We consider pairs $(\rho, \rho') \in TA$ along with minimal ν, ν' such that $(\rho\nu, \rho'\nu') \in TA$ (e.g. for the trace alignment in Figure 5.3, we consider the pairs shown in Table 5.1). For each such pair we add a transition $(p, p') \rightarrow (q, q')$ labeled by the paths of basic blocks taken by ν and ν', where (p, p') is the last pair of program points in ρ, ρ' and (q, q') is the last pair of program points in ν, ν'. As an optimization, we consider only ν, ν' that are small, for example, fewer than 10 machine states in length.

The PAA can be regarded as a nondeterministic finite automaton (NFA) in the following sense. A pair of traces ρ, ρ' for f and g is *accepted* by the PAA if there is a series of transitions (a *run* through the PAA) from the start node to the exit node that correspond with ρ, ρ' (i.e. concatenating the labels of the transitions gives paths that match paths taken by ρ and ρ'). The above construction ensures that every pair of traces in the training set is accepted by the PAA (we say the PAA "accepts the training set").

After performing this construction, we simplify the PAA by removing nodes and transitions while ensuring the PAA still accepts the training set. Removing nodes makes finding provably correct invariants easier, and removing transitions decreases the number of proof obligations. In our experiments, we find simplification reduces the number of nodes by 3.9x and the number of edges by 3.7x. We perform the following two operations until we reach a fixedpoint.

First, we remove every node s that does not have a self-loop other than the entry and exit. Suppose s has incoming transitions $r_1 \rightarrow s, \ldots, r_n \rightarrow s$ and outgoing



(a) PAA before simplification.

(b) After removing node q_2q_1'.

(c) After deleting the extra edge.

Figure 5.7: Example of simplification procedure.

transitions $s \to t_1, \ldots, s \to t_m$. For each i, j-pair, replace the transitions $r_i \to s$ and $s \to t_j$ with a transition $r_i \to t_j$, labeled with the concatenation of the paths of the original two transitions.

Second, we remove extra transitions. If a transition $\lambda : s \to t$ is labeled with paths P, Q and transition $\lambda^* : s \to u$ is labeled with paths P^*, Q^* where P is a prefix of P^* and Q is a prefix of Q^*, then λ^* is removed. Once we cannot remove any more transitions or nodes, the PAA is simplified.

Figure 5.7a shows a hypothetical PAA for the example (Section 5.2). We can remove node q_2q_1' since it has no self loops. We combine the transition $a;\epsilon$ with each of $b;a'$ and $bbb;a'c'$ to get two transitions $q_1q_1' \to q_2q_2'$, as shown in Figure 5.7b. Because ab is a prefix of $abbb$ and a' is a prefix of $a'c'$, we can remove one more transition to obtain the simplified PAA shown in Figure 5.7c.

5.4.3 Testing the Program Alignment Automaton

If the alignment predicate is chosen poorly, the PAA constructed in Section 5.4.2 may be overfitted to the training data. As a worst case example, consider the alignment predicate $\xi =$ "false". Traces will only be aligned by ξ at the beginning and the end. Every new test case may add a new transition from the start node to the end node labeled by the entire pair of traces. Thus, there is no limit on the number of transitions (if the traces can be arbitrarily long) and such a PAA is not useful for equivalence checking. A good alignment predicate, on the other hand, results in a PAA to which no further transitions need to be added to accept additional test cases — the PAA already captures all possible pairs of executions of the two programs. We must verify that the PAA is such a sound overapproximation of the two programs as part of equivalence checking (see Section 5.4.5), but we can eliminate many PAAs earlier by testing. We use a separate *test set* of inputs for this purpose.

By construction, the PAA accepts the training set (Section 5.4.2), so we check that the PAA also accepts the test set. This check is similar to the standard language membership test for NFAs. If the PAA fails to accept the test set, then we reject the PAA and try another alignment predicate.

5.4.4 Learning Invariants

Our goal is to learn invariants for each node of the PAA. We take a data-driven approach and use the test cases to guess a conjunction of predicates for each node, and later (Section 5.4.5), we discard the conjuncts that cannot be proven.

First, for each node s of the PAA, we need to identify a set of pairs of machine states, Σ_s, over which to learn invariants. For the traces of each test case (from either the test set or the training set), we consider every possible run of the PAA and record the machine states at each transition. Given a test case τ, we run both programs to obtain traces ρ, ρ'. Let ν, ν' be prefixes of ρ, ρ' that execute paths P, Q. Consider every sequence of transitions (if any) from the start node of the PAA such that the concatenation of the labels of the transitions match P and Q. For each such sequence of transitions that ends in node s, we add the pair (σ, σ') to Σ_s where σ and σ' are the last machine states of ν and ν'.

The language of invariants, shown in Figure 5.8, includes linear equalities, inequalities and equalities mod n. Inequalities are needed for reasoning about branch conditions, and equalities mod n are needed to prove properties of warm-up and cool-down loops in vectorized code. There are three different data-driven techniques

$$\begin{aligned} Inv \;\; \rightarrow \;\; & \textstyle\sum c_i v_i = c \;\big|\; v_1 - v_2 \leq c \;\big|\; \pm\, v \leq c \\ & \big|\; m = v \;\big|\; v_1 - v_2 \equiv c_1 \pmod{c_2} \\ & \big|\; v \equiv c_1 \pmod{c_2} \;\big|\; \omega_{\overline{S}} = \omega'_{\overline{S}} \end{aligned}$$

Figure 5.8: The language of invariants. Each c represents a bitvector constant and each m represents a memory location. v represents a register, a subregister, or a stack-allocated memory location. $\omega_{\overline{S}}$ represents the heap excluding the set of memory locations S.

for learning the conjuncts.

First, for inequalities, we sample a subset of the data and find all the inequalities with the strongest bound possible. Then, we check if these inequalities hold over the entire data set; the failing ones are discarded.

Second, we use techniques from linear algebra to find a space of all linear equalities that hold over the data (see Section 2.3). We construct a matrix M over the ring of 64-bit bitvectors $\mathbb{Z}_{2^{64}}$ containing program values, where row i corresponds to σ_i and σ'_i, and column j corresponds to a register or stack location. We use SageMath version 7.5.1 [63] to compute the kernel $K = \ker M$. Each vector in the generating set for K corresponds to a linear equality that holds over all pairs (σ_i, σ'_i). Performing this computation over $\mathbb{Z}_{2^{64}}$ rather than $\mathbb{Z}$ is expensive, but necessary because some equalities hold over $\mathbb{Z}_{2^{64}}$ that do not hold over $\mathbb{Z}$ (in past work [59, 14], the invariant learning routine would miss some of these equalities). Therefore, we perform two optimizations. First, we do a pre-pass in which we remove pairs of columns where a linear relationship of the form $c_1v_1 + c_2v_2 = 0$ can be readily found. Second, we only sample $k = 25$ rows of test data for the matrix. We test the learned invariants against the rest of the data set; if the learned invariants do not hold, we sample up to k more rows from among the failures and repeat.

Lastly, for the remaining classes of invariants, we learn the strongest possible invariant over the entire data set. Here, no division between test and training data is needed. We attempt to learn an equality mod n for every pair of program values. For each pair v_1, v_2 we compute all differences $d_i = v_1 - v_2'$ for each σ_i, σ_i'. We then compute the greatest common divisor d of all $d_i - d_j$. If $d \neq 1$, then we can find c such that $v_1 - v_2 \equiv c \pmod{d}$. To learn an invariant of the form $\omega_{\overline{S}} = \omega'_{\overline{S}}$, we choose a minimal set S for which the invariant holds on all test cases.

Invariants learned for the PAA in Figure 5.4 are shown in Figure 5.5.

5.4.5 Verifying Proof Obligations

We perform a Houdini-style [27] fixedpoint computation to reduce the set of learned invariants to those that can be proven by induction. For each node s we have the invariant $\phi_s = \phi_s^1 \wedge \cdots \wedge \phi_s^n$. For each transition $\lambda : t \to s$ labeled by paths P and Q we attempt to prove $\{\phi_t\}\ P\ ;\ Q\ \{\phi_s^i\}$ for $1 \leq i \leq n$. If any conjunct does not hold, we remove it from the invariant. We repeat this procedure until all the proofs succeed. We then check the remaining proof obligations (Section 5.3.1), and Corollary 2 implies equivalence.

Our implementation supports two ways to model the stack. The first models the stack conservatively, where we assume that the stack pointer is an arbitrary address that could alias with arbitrary data structures on the heap, and ensures that the two functions behave identically. However, for verifying optimizations that transform the stack, we also support assuming that stack locations do not alias with any heap locations or pointers in input parameters. In these cases we also assume that stack accesses of different sizes do not alias, so we model them using separate memory stores [66].

5.4.6 Space of Alignment Predicates

In practice, we find there is a small space of predicates that almost always contains a useful alignment predicate for pairs of equivalent `x86-64` functions. Namely, choosing a predicate of the form $(c_1 v_1 - c_2 v_2 = k) \wedge \omega = \omega'$ is typically sufficient. Here, v_1 and v_2

are registers or stack-allocated locations in f and g. We restrict $c_1, c_2 \in \{1, 2, 4, 8, 16\}$ and $k \in \mathbb{Z}$. Moreover, we only need to consider alignment predicates where either $c_1 = 1$ or $c_2 = 1$. There are 16 registers, but we only need to consider registers whose values are defined (as determined by a program analysis). Thus, the total number of choices for c_1, c_2, v_1 and v_2 has a relatively small bound. For each of these, we heuristically pick k by finding values seen for $c_1v_1 - c_2v_2$ across different states that are in common across multiple pairs of traces. Performing this search is generally quite fast, and we make no attempt to rank the alignment predicates heuristically or try them in a particular order. If all the alignment predicates fail, we additionally attempt to use predicates of the form $c_1v_1 - c_2v_2 = k$, where the alignment predicate does not constrain the heap states. We offer intuition for, and evaluate the utility of, this space of alignment predicates in Section 5.5.5.

5.5 Evaluation

In this section we seek to validate the following points:

- Our technique is able to verify the correctness of vectorization and other complex loop transformations as performed by modern compilers on `x86-64`. (Sections 5.5.1 and 5.5.2)
- Our technique can verify optimizations that are beyond the scope of existing automated black-box techniques. (Section 5.5.3)
- Our technique can verify equivalence checking benchmarks used to evaluate other state-of-the-art tools. (Section 5.5.4)
- The search space of alignment predicates that we use is suitable for realistic verification problems. (Section 5.5.5)
- Alias relationship mining is helpful for discharging proof obligations. (Section 5.5.6)

We conclude with limitations in Section 5.5.7.

5.5.1 Experimental Setup

To evaluate our method, we construct a set of benchmarks for verifying vectorization optimizations. We started with 156 functions from the *Test Suite for Vectorizing Compilers* (TSVC), which was developed "to assess the vectorizing capabilities of compilers" and ported to C in [41]. We removed five classes of functions from the original TSVC set:

- Functions that could not be vectorized using `-msse4.2` and `-O3` with either `gcc` 4.9.2 or `clang` 3.4. These functions are not interesting in our evaluation because the loop structures are preserved. These functions should be easy for both our technique and other state-of-the-art tools. (96 functions)

- Duplicate functions. Some TSVC functions were designed to check that a compiler could perform an analysis to verify the safety of an optimization; however, after successful vectorization, the generated `x86-64` code matches that of another function. (6 functions)
- Functions with method calls. Our implementation does not support method invocations. (9 functions)
- Out-of-scope functions designed to test loop interchange. See Section 5.5.7. (6 functions)
- Functions with two-dimensional arrays or memory indirection. (11 functions)

The TSVC functions operate on statically-allocated, fixed-size global arrays of floating point values. While our technique works as is on over 80% of the floating point benchmarks (by using uninterpreted functions to model floating point operations), there are additional issues when learning invariants from floating point data that are not addressed by existing invariant inference techniques. For example, there are multiple binary representations for some floating point values, such as `NaN`. These issues are orthogonal to our contributions; to separate the evaluation of our method from the details of floating point semantics, we systematically replaced floating point types with integer types. We constructed 256 test cases by creating machine states containing input arrays of randomly-chosen bytes. This same set of test cases was sufficient to obtain code coverage over all these benchmarks. We also added a parameter to each function to specify the array length. We added assumptions on the input values to prevent pointers for different arrays from aliasing. Adding assumptions to avoid undefined behavior is generally required for equivalence checking [18, 60].

We were left with 28 functions. Most iterate over one or more arrays (up to 5), perform arithmetic, and update the arrays. Some process the array forwards, some backwards, and some with a stride. Some have loop-carried dependencies, others do not. One function, `s176`, features a doubly-nested loop. No combination of these features hindered our ability to check equivalence. For each of the 28 functions we attempted to prove that `gcc -O1` code was equivalent to `gcc -O3` code and to `clang -O3` code, resulting in a total of 56 benchmarks.

Sometimes, discharging a particular proof obligation takes a long time or times out using one SMT solver, but finishes quickly with another solver. Thus, we use two solvers, `Z3` [21] (commit `7f6ef0b6`) and `CVC4-1.5` [6], with the theories of arrays and bitvectors. Also, the encoding of constraints that represent memory accesses may have a profound impact on solver performance. Therefore we implement two memory models [66], a flat memory model, and one based on alias relationship mining (ARM, see Chapter 3). The flat model encodes all memory accesses as a read or an update to an array (with separate arrays for the stack, if needed). ARM uses data from test cases to guess and prove relationships that ensure pointers do not alias; then the constraints are encoded with minimal use of arrays [14]. We set a 30-minute timeout for each proof obligation for each solver and memory model. We use the result of whichever solver and memory model pair finishes first. We use the counterexamples from the SMT solver to eliminate other proof obligations that are demonstrably false, as in [30].

We used rigorously tested semantic models for `x86-64` instructions developed by hand [14] and synthesized automatically [32]. We model multiplications and floating point operations using uninterpreted functions. We performed the construction of the PAA for each benchmark using one core of an Intel Xeon E5-2667 CPU @ 3.3GHz machine. We use a pool of preemptible cloud virtual machines to check the proof obligations.

5.5.2 Results

A list of the benchmarks and the outcomes are shown in Table 5.2. We successfully verified 55 of the 56 benchmarks. The one failure (`s351-gcc`) was due to a timeout. In all other cases the proofs succeeded. The PAAs all had 3 or 4 nodes. The number of edges varied from 4 to 254, with a median of 4 and an average of 9. The number

	`gcc -O1`	`gcc -O3`		`clang -O3`	
Benchmark	LOC	LOC	Out	LOC	Out
s000	14	18	✓	44	✓
s1112	12	31	✓	59	✓
s112	14	55	✓	24	✓NV
s121	16	44	✓	48	✓
s1221	14	24	✓	37	✓
s122	17	108	✓S	21	✓NV
s1251	18	29	✓	60	✓
s127	22	82	✓	31	✓
s1281	21	30	✓	66	✓
s1351	12	17	✓	51	✓
s162	17	49	✓	58	✓
s173	17	56	✓	70	✓S
s176	29	99	✓S	34	✓NV
s2244	19	56	✓	65	✓
s243	25	30	✓NV	68	✓
s251	16	27	✓	49	✓
s3251	22	149	✓S	26	✓NV
s351	29	130	×S	24	✓
s452	22	27	✓	25	✓
s453	15	22	✓	15	✓NV
sum1d	15	28	✓	45	✓
vdotr	17	28	✓	49	✓
vpvpv	15	26	✓	38	✓
vpv	13	25	✓	37	✓
vpvts	14	30	✓S	54	✓
vpvtv	14	26	✓	36	✓
vtv	14	25	✓	36	✓
vtvtv	15	26	✓	51	✓

Table 5.2: Results for 56 vectorization benchmarks. ✓ represents successful verification and × represents a timeout. For six functions, only one compiler succeeds in vectorization; these benchmarks are marked by NV. Benchmarks requiring assumptions about the stack are marked by S.

of conjuncts in the invariants in the final PAA ranged from 374 to 1417, with a median of 651. The median time to discharge all proof obligations was 45.0 CPU hours; the minimum time was 2.5 CPU hours (`s112-clang`) and the maximum 1166 CPU hours (`s351-clang`). The end-to-end time for this benchmark using the cloud was 4.6 hours. The cost for cloud instances was $0.01 per CPU hour, so the cost of checking the proof obligations for this benchmark was $11.66 while a typical problem cost just $0.45.

The most difficult benchmark, `s351`, includes a loop with five multiplications, five additions, and ten memory dereferences in each iteration. The `s351-gcc` benchmark, which encountered timeouts while checking proof obligations, included a 4-way vectorized loop with a fully-unrolled cool-down loop to handle the last four iterations. It is likely that with more effort, our constraint generation procedure can be tuned to discharge the problematic proof obligations more efficiently. While `s351-clang` still used vector instructions, `clang` generated much simpler code than `gcc`. Still, the `s351-clang` benchmark took the most CPU time of all the successful benchmarks.

5.5.3 GNU C Library `strlen` Case Study

Sometimes compilers are unable to vectorize performance-critical functions, and so library developers perform the vectorization themselves. This is the case for the `strlen` function in `libc`, which was most recently updated in May 2009 with the release of version 2.10.1. There is a test in the `libc` test suite that runs both a reference implementation and the hand optimized one, and checks that the outputs are equal. Instead of running the programs on some inputs, we can leverage test cases to prove that the two implementations are equivalent for all inputs. We successfully verified the correctness of the `strlen` function (shown in Figure 5.9a) originally released in 2.10.1, which still ships as of 2019 in version 2.29, against a simple reference implementation (Figure 5.9b). The alignment predicate found asserts the equality of the pointers into the string (`ptr` and `p`). The end-to-end verification time was only 3.3 minutes on a single CPU core.

```
size_t strlen (char *str) {
  char *ptr;
  ulong *longword_ptr;
  ulong longword, himagic, lomagic;

  for (ptr = str; ((ulong) ptr & 7) != 0; ++ptr)
    if (*ptr == '\0')
      return ptr - str;

  longword_ptr = (ulong *) ptr;
  himagic = 0x8080808080808080L;
  lomagic = 0x0101010101010101L;

  for (;;)
  {
    longword = *longword_ptr++;
    if ((longword - lomagic) & ~longword & himagic)
    {
      char *cp = (char*)(longword_ptr - 1);
      if (cp[0] == 0) return cp - str;
      if (cp[1] == 0) return cp - str + 1;
      if (cp[2] == 0) return cp - str + 2;
      if (cp[3] == 0) return cp - str + 3;
      if (cp[4] == 0) return cp - str + 4;
      if (cp[5] == 0) return cp - str + 5;
      if (cp[6] == 0) return cp - str + 6;
      if (cp[7] == 0) return cp - str + 7;
    }
  }
}
```

(a) Vectorized `strlen` implementation (simplified). The main loop has eight different branches to exit, and the warm-up loop has two. Compilation adds an extra branch that skips the warm-up loop. The alignment predicate ensures that each of these paths is mapped to the correct number of iterations in the reference implementation.

```
size_t strlen (char *s) {
  char* p;
  for(p = s; *p; ++p);
  return p - s;
}
```

(b) Reference `strlen` implementation.

Figure 5.9: Two implementations of `strlen`.

The vectorized code has two loops; the warm-up loop (lines 6-8) counts characters one-by-one until the pointer reaches an 8-byte boundary or a null character. The main loop (lines 14-29) reads 8 characters from the string at a time and uses clever bit-manipulation techniques to check if any of the 8 characters are null. If so, the code checks the remaining characters one-by-one and returns the length; otherwise, the loop continues.

Thus, the code reads beyond the end of the string unless the string ends at an 8-byte boundary or the warm-up loop encounters the null terminator. This is safe on `x86-64` because memory permissions are set on a page-level granularity (usually 4kB in size). If m is a memory address the process is allowed to read, so is $8\lfloor m/8 \rfloor + 7$. While the optimized code can perform an out of bounds read, it never uses this value, and the read does not trigger a page fault (assuming that the unoptimized code does not fault). This example shows two programs that are provably equivalent, even though they dereference a different set of memory locations.

In general, if the memory locations accessed by f are provably on the same pages as those accessed by g, then f raises a page fault if and only if g does; but, if the memory accesses are on different pages, no such guarantee exists. For the sake of checking aggressive optimizations, we decided not to model page faults (we do, however, check for final heap equality, which addresses most faults due to memory writes). Thus $\{\phi_1\}\ P\ ;\ Q\ \{\phi_2\}$ may hold, even if path P contains a memory access but path Q does not. There is no guarantee that equivalent programs will access memory pages in the same order; f could read and write a memory location in each loop iteration, while g reads and writes the memory location once (Section 5.5.4 offers one such example). Therefore, fully modeling page faults likely requires invariants that track which memory locations each program has accessed.

We also discovered that the hand-optimized code was written conservatively. When the guard of the if-statement on line 17 is satisfied, one of the eight return statements is always taken. We can optimize the code by moving the cascade of if-statements to the outside of the loop, and we proved this is sound.

To the best of our knowledge, no other black-box technique in the equivalence checking, relational verification, or translation validation literature is able to automatically verify this example. There are two challenging aspects to highlight. First, the number of iterations executed in the warm-up loop and main loops are data-dependent; i.e., the number of iterations of the warm-up loop depends on the alignment of the input string to an 8-byte boundary. Second, the PAA has a large number of edges, and a naive search to build the PAA is too inefficient. Using an alignment predicate makes the search tractable.

5.5.4 Comparison with Related Work

We believe techniques that depend on syntactic alignment of the two programs [59, 45, 48, 25, 26] fail on most or all of our benchmarks, including at least 47 benchmarks where loop unrolling has been performed (usually as part of vectorization). In [8], the authors suggest unrolling one loop and then attempting a syntactic alignment. This approach does not support cool-down loops (present in 21 of our benchmarks) or loop peeling optimizations (present in another 9 benchmarks). The technique of [17] succeeds on benchmarks unrolled μ times, where μ is an unroll factor. The cost of the technique is superexponential in μ in the worst case, and reported results are only for $\mu = 1$ [17, 30]. Among our benchmarks, 32 have been unrolled 4 times and 15 have been unrolled 8 times. Finally, we believe ours is the only black box, automated technique able to check equivalence for `libc strlen` (Section 5.5.3) and our running example (Section 5.2).

A challenging equivalence checking problem is presented in [17]. As far as we know, only our technique and the technique of [17] are able to handle this problem. The benchmark consists of checking the correctness of a loop that sums the positive integers of an array after optimizations have been performed, including loop inversion, a transformation of branch conditions, replacing a branch inside the loop with a conditional move instruction, and register allocation. The unoptimized program writes to a global heap variable on every iteration, while the optimized version only writes the result once at the end of the loop. Their benchmark was for 32-bit `x86` rather than `x86-64`, but we found that compiling the C source on `x86-64` with `gcc 4.9.2` using `-O0` and `-O1` produced the same control flow graphs and the same optimizations; we believe that this modified benchmark is a suitable proxy for the original.

We successfully verified this benchmark; the alignment predicate we found related the stack-allocated pointer of the unoptimized program with an index counter in the optimized one and did not relate heap states. The total time to guess the alignment predicate, construct the PAA, learn invariants, and verify the proof obligations on a single CPU core was 34.4 minutes. Most of the time was spent verifying the proof obligations, which was done only using Z3 and only with the flat memory model.

The authors of [17] also demonstrate a large-scale evaluation of their technique on whole binaries, but in whole programs many of the equivalence checks between corresponding functions are easy (e.g. do not involve loop optimizations), and [17] does not describe the harder equivalence checking problems. Since our contribution is about equivalence checking of loops, our evaluation focuses on loops rather than whole programs.

5.5.5 Search over Alignment Predicates

We performed an experiment to count the number of alignment predicates in the search space for each benchmark, and the number of viable PAAs that we could build (meaning the number of PAAs that accept the test set; see Section 5.4.3). For each benchmark we tried between 182 and 3318 alignment predicates, with a median of 1130. Between 0.37% and 22% of these alignment predicates led to viable PAAs. Averaging across the benchmarks, 3.1% of alignment predicates succeeded. At least 8 viable PAAs were found per benchmark, with a maximum of 65 and a median of 28. These findings suggest that our space of alignment predicates is robust for our set of benchmarks.

In practice the successful alignment predicates typically relate a pointer or counter in f with a pointer or counter in g. This is the case in our example (Section 5.2), where the alignment predicate matches the value of a pointer in f, namely `array + 4i`, with the pointer `array'` in g. When g processes 8 bytes of the array and the pointer increases by 8, we find a corresponding path in f where 8 bytes are processed and its pointer increases by 8. The powers of two in our alignment predicates arise because counters are generally multiplied by powers of two to address array locations, and not due to specifics of any optimizations performed on our benchmarks (e.g. the number of loop iterations unrolled). Alternatively, for the example, we can use equality of heap states as the alignment predicate to ensure that the memory writes of f and g are aligned and obtain the same result. For benchmarks where heap equality alone was a suitable alignment predicate, a large proportion of alignment predicates worked. We also observe that some alignment predicates succeed in aligning one loop an iteration (or k iterations) ahead of the other. We can check the proof obligations for the resulting PAAs as long as the invariants are able to sufficiently relate the program states despite this offset.

Since there are only a few ways to reference a memory location on `x86-64`, it is unsurprising that even our simple alignment predicates suffice to identify corresponding uses of pointers and counters between the two programs. For example, if f accesses an array using a pointer in register r_1, and g accesses an array of k-byte elements using a base address b and counter register r_2, then the alignment predicate $r_1 = b + k * r_2$ would assert the equality of these two memory dereferences. Indeed, this is in our space of alignment predicates.

While it did not arise in our benchmarks, we expect that some equivalence checking problems will require the alignment predicate to assert an equality over three or four registers. Four registers would be the maximum required to relate any two pointer dereferences. We also expect that some benchmarks involving multiple loops will require a disjunction over program points; for example, we may want to use one alignment predicate for one loop, and another alignment predicate for another loop. While one could extend our work to such problems by broadening the space of alignment predicates and thus increasing search times, our observation that good alignment predicates tend to relate pointers and counters suggests that these alignment predicates may be guessed directly from the program text. We leave this question to future work.

5.5.6 Evaluation of Memory Models

As described in Chapter 3 of this thesis, alias relationship mining (ARM) is a technique that improves the reliability of discharging proof obligations with an SMT solver in the presence of memory references.

In our experiments, we found that the majority of the benchmarks could be discharged using only a single SMT solver and the flat memory model. Beyond these, even more benchmarks can be discharged using the flat memory model along with two SMT solvers running in parallel (once one finishes we stop the second). However, among the benchmarks, there are a few for which ARM is truly required. Moreover, the only way to discover which combination of solver and memory model is most suited for a benchmark is to actually run the benchmark – hence we run the different solver and memory model combinations in parallel.

Table 5.3 shows, for the `s452-llvm` benchmark, the number of proof obligations that were successfully discharged using a 30-minute timeout with the four solver and memory model pairs alone. Then, we show the number that may be discharged using a single SMT solver (Z3 or CVC4), but trying both of the memory models in parallel. We also do the opposite: choose a fixed memory model, then run both SMT solvers in parallel, and see how many proof obligations can be discharged. Lastly, we run all four solver and memory model combinations and count the number of proof obligations which do not suffer timeouts. For this particular benchmark, we see that CVC4 succeeded on all the problems that Z3 did, and that CVC4 alone would be sufficient (while Z3 only succeeded on 78% of those CVC4 completed). However, neither the flat memory model nor alias relationship mining with CVC4 were sufficient to discharge the proof obligations; the flat memory model succeeded

	Z3	CVC4	Z3+CVC4
Flat	1399	1788	1788
ARM	1394	1796	1796
Flat+ARM	1399	1800	1800

Table 5.3: Number of proof obligations discharged by different solvers and memory models for benchmark `s452-llvm`.

in checking 1788 of them, while ARM succeeded on 1796. However, there were 4 proof obligations for which the flat memory model succeeded where ARM failed; both memory models are needed to verify all 1800 of the proof obligations. Using CVC4 with ARM is nearly, but not quite, sufficient for this example. With more engineering, all the obligations could likely be discharged using a single memory model; even so, we expect that trying different combinations of solvers and memory models will still be useful for challenging benchmarks.

5.5.7 Limitations

A main limitation of our work is that we cannot reason about transformations that reorder an unbounded number of memory writes, for example, loop splitting, loop fusion, loop interchange, and loop tiling optimizations. This is because the only invariants we learn and prove over the heap states assert heap equality on all but a finite set of memory locations. This limitation could be addressed by learning and proving more general quantified invariants over heap states.

Another limitation arises when the correspondence between the control flow of the two programs depends on an unbounded input. Consider the two functions in Figure 5.10 where the loop of f has been flattened. Here, m iterations in f correspond to 1 iteration in g. As far as we know, no equivalence checking techniques that construct a product program or similar structures are able to verify this benchmark as is (although those that summarize loops, like [20], may succeed). The reason is that the product program needs to align the entire execution of the inner loop of f

```
int f(uint n, uint m) {
  int k = 0;
  for(uint i = 0; i < n; ++i) {
    for(uint j = 0; j < m; ++j) {
      k++;
    }
  }
  return k;
}
int g(uint n, uint m) {
  int k = 0;
  for(uint i = 0; i < n; ++i) {
    k += m;
  }
  return k;
}
```

Figure 5.10: A difficult problem for equivalence checking via product programs.

with m iterations of the loop of g. To extend our approach to benchmarks like these, we would need to summarize loops (in this case the inner loop of f) and check for termination.

However, we confirmed our method can verify a modified version of this benchmark where other approaches using product programs likely fail. If the input value m is constrained to a small finite set $m \in \{c_1, c_2, \ldots, c_k\}$ while n is left unbounded, then we can construct a PAA for the two programs and prove equivalence. The PAA contains a node s with k transitions $\lambda_i : s \to s$ where λ_i relates c_i iterations of f to 1 iteration of g. In essence, the PAA we learn creates a disjunction of all the k cases and we check each one. We can reason disjunctively because the path condition for λ_i only holds when $m = c_i$.

Chapter 6

Conclusion

In this thesis, we described three contributions to the equivalence checking literature. First, in Chapter 3, we described how alias relationship mining may help discharge proof obligations. In Chapter 4, we gave an application of equivalence checking to loop superoptimization; we showed that a bounded validation technique, in addition to a sound one, is necessary for successfully superoptimizing loops soundly. Lastly, we described and evaluated a new technique – semantic program alignment – which can be used to verify the correctness of aggressive compiler optimizations that alter control flow.

Bibliography

[1] Chrome rewards. `https://www.google.com/about/appsecurity/chrome-rewards/`. Accessed: Aug 2016.

[2] Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D. Evaluating the "small scope hypothesis". In *Principles of Programming Languages* (2002), POPL '02.

[3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A transparent dynamic optimization system. In *Programming Language Design and Implementation* (2000), PLDI '00.

[4] Balakrishnan, G., and Reps, T. W. WYSINWYX: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems 32*, 6 (2010).

[5] Bao, W., Krishnamoorthy, S., Pouchet, L.-N., Rastello, F., and Sadayappan, P. Polycheck: Dynamic verification of iteration space transformations on affine programs. In *Principles of Programming Languages* (2016), POPL '16, pp. 539–554.

[6] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Computer Aided Verification* (2011), vol. 6806 of *CAV '11*, pp. 171–177.

[7] Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., and Zuck, L. Tvoc: A translation validator for optimizing compilers. In *Computer Aided Verification* (2005), CAV '05.

[8] Barthe, G., Crespo, J. M., Gulwani, S., Kunz, C., and Marron, M. From relational verification to SIMD loop synthesis. In *Symposium on Principles and Practice of Parallel Programming* (2013), PPoPP '13, pp. 123–134.

[9] Barthe, G., Crespo, J. M., and Kunz, C. Relational verification using product programs. In *International Conference on Formal Methods* (2011), FM '11, pp. 200–214.

[10] Benton, N. Simple relational correctness proofs for static analyses and program transformations. In *Principles of Programming Languages* (2004), POPL '04, pp. 14–25.

[11] Burstall, R. M. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7* (1972).

[12] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In *Symposium on Operating Systems Principles* (2009), SOSP '09.

[13] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S. B., and Yates, J. Fx! 32: A profile-directed binary translator. *IEEE Micro 18*, 2 (1998).

[14] Churchill, B., Sharma, R., Bastien, J., and Aiken, A. Sound loop superoptimization for google native client. In *Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 313–326.

[15] Csallner, C., Tillmann, N., and Smaragdakis, Y. Dysy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering* (2008), ICSE '08, pp. 281–290.

[16] Currie, D., Feng, X., Fujita, M., Hu, A. J., Kwan, M., and Rajan, S. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming 32*, 3 (2006).

[17] Dahiya, M., and Bansal, S. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programing Languages and Systems* (2017), ASPLAS '17, pp. 127–147.

[18] Dahiya, M., and Bansal, S. Modeling undefined behavior semantics for checking equivalence across compiler optimizations. In *Haifa Verification Conference* (2017), HVC '17.

[19] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Relational verification through horn clause transformation. In *International Static Analysis Symposium* (2016), SAS '16, pp. 147–169.

[20] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Enhancing predicate pairing with abstraction for relational verification. *arXiv preprint arXiv:1709.04809* (2017).

[21] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Theory and Practice of Software, Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS '08.

[22] Dutta, S., Sarkar, D., Rawat, A., and Singh, K. Validation of loop parallelization and loop vectorization transformations. In *Evaluation of Novel Software Approaches to Software Engineering* (2016), ENASE 2016, pp. 195–202.

[23] Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T. Abstract domains of affine relations. *ACM Transactions on Programming Languages and Systems 36*, 4 (2014).

[24] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1–3 (2007).

[25] Fedyukovich, G., Gurfinkel, A., and Sharygina, N. Automated discovery of simulation between programs. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2015), pp. 606–621.

[26] Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., and Ulbrich, M. Automating regression verification. In *Automated Software Engineering* (2014), ASE '14, pp. 349–360.

[27] Flanagan, C., and Leino, K. R. M. Houdini: An annotation assistant for ESC/Java. In *Formal Methods Europe* (2001), FME '01, pp. 500–517.

[28] Friedberg, S. H., Insel, A. J., and Spence, L. E. *Linear Algebra*, fourth edition ed. Pearson Education, Upper Saddle River, New Jersey, 2003.

[29] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. Synthesis of loop-free programs. In *Programming Language Design and Implementation, (PLDI)* (2011).

[30] Gupta, S., Saxena, A., Mahajan, A., and Bansal, S. Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In *Theory and Applications of Satisfiability Testing* (2018), SAT '18.

[31] Hawblitzel, C., Lahiri, S. K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., and Wadsworth, S. Will you still compile me

tomorrow? Static cross-version compiler validation. In *Foundations of Software Engineering* (2013), ESEC/FSE '13, pp. 191–201.

[32] Heule, S., Schkufza, E., Sharma, R., and Aiken, A. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Programming Language Design and Implementation* (June 2016), PLDI '16.

[33] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[34] Jackson, D., and Damon, C. A. Elements of Style: Analyzing a software design feature with a counterexample detector. In *Software Testing and Analysis* (1996), ISSTA '96.

[35] Joshi, R., Nelson, G., and Zhou, Y. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems 28*, 6 (2006).

[36] Kanade, A., Sanyal, A., and Khedker, U. P. A PVS based framework for validating compiler optimizations. *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)* (2006), 108–117.

[37] Kiefer, M., Klebanov, V., and Ulbrich, M. Relational program reasoning using compiler IR. In *Working Conference on Verified Software: Theories, Tools, and Experiments* (2016), pp. 149–165.

[38] Kundu, S., Tatlock, Z., and Lerner, S. Proving optimizations correct using parameterized program equivalence. In *Programming Language Design and Implementation* (2009), PLDI '09, pp. 327–337.

[39] Leroy, X. A formally verified compiler back-end. *Journal of Automated Reasoning 43*, 4 (2009).

[40] Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. Provably correct peephole optimizations with alive. In *Programming Language Design and Implementation* (2015), PLDI '15, pp. 22–32.

[41] Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., and Padua, D. A. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques* (2011), PACT '11, pp. 372–382.

[42] Mangpo, P., Thakur, A., Bodik, R., and Dhurjati, D. Scaling up superoptimization. In *Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16.

[43] Mao, O., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F. Software fault isolation with API integrity and multi-principal modules. In *Symposium on Operating Systems Principles* (2011), SOSP '11.

[44] Massalin, H. Superoptimizer: A look at the smallest program. In *Architectual Support for Programming Languages and Operating Systems* (1987), ASPLOS '87, pp. 122–126.

[45] Necula, G. C. Translation validation for an optimizing compiler. *ACM Sigplan Notices 35*, 5 (2000).

[46] Nimmer, J. W., and Ernst, M. D. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science 55*, 2 (2001), 255 – 276. Runtime Verification.

[47] Oetsch, J., Prischink, M., Pührer, J., Schwengerer, M., and Tompits, H. On the small-scope hypothesis for testing answer-set programs. In *Principles of Knowledge Representation and Reasoning* (2012).

[48] Partush, N., and Yahav, E. Abstract semantic differencing for numerical programs. In *Static Analysis Symposium* (2013), SAS '13, pp. 238–258.

[49] Person, S., Dwyer, M. B., Elbaum, S. G., and Pasareanu, C. S. Differential symbolic execution. In *Foundations of Software Engineering* (2008), FSE '08.

[50] Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems* (1998), TACAS '98, pp. 151–166.

[51] Ramos, D. A., and Engler, D. R. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification* (2011), CAV '11.

[52] Rival, X. Symbolic transfer function-based approaches to certified compilation. In *Principles of Programing Languages* (2004), POPL '04, pp. 1–13.

[53] Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. *SIGPLAN Not. 48*, 4 (2013), 305–316.

[54] Schkufza, E., Sharma, R., and Aiken, A. Stochastic optimization of floating-point programs with tunable precision. In *Programming Language Design and Implementation* (2014), PLDI '14.

[55] Schkufza, E., Sharma, R., and Aiken, A. Stochastic program optimization. *Communications of the ACM 59*, 2 (Jan. 2016), 114–122.

[56] Sehr, D., Muth, R., Biffle, C. L., Khimenko, V., Pasko, E., Yee, B., Schimpf, K., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium* (2010).

[57] Sewell, T. A. L., Myreen, M. O., and Klein, G. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation* (2013), PLDI '13, pp. 471–482.

[58] Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., and Nori, A. V. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems* (2013), pp. 574–592.

[59] Sharma, R., Schkufza, E., Churchill, B., and Aiken, A. Data-driven equivalence checking. In *Object Oriented Programming Systems Languages and Applications* (2013), OOPSLA '13, pp. 391–406.

[60] Sharma, R., Schkufza, E., Churchill, B., and Aiken, A. Conditionally correct superoptimization. In *Object-Oriented Programming, Systems, Languages, and Applications* (2015), OOPSLA '15, pp. 147–162.

[61] Stepp, M., Tate, R., and Lerner, S. Equality-based translation validator for LLVM. In *Computer Aided Verification* (2011), CAV'11, pp. 737–742.

[62] Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. Equality saturation: A new approach to optimization. In *Principles of Programming Languages* (2009), POPL '09, pp. 264–276.

[63] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.5.1)*, 2017. `http://www.sagemath.org`.

[64] Tristan, J.-B., Govereau, P., and Morrisett, G. Evaluating value-graph translation validation for LLVM. In *Programming Language Design and Implementation* (2011), PLDI '11, pp. 295–305.

[65] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Operating Systems Review 27*, 5 (1994).

[66] Wang, W., Barrett, C., and Wies, T. Partitioned memory models for program analysis. In *Verification, Model Checking, and Abstract Interpretation* (2017), VMCAI '17, pp. 539–558.

[67] Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland)* (2009).

[68] Zaks, A., and Pnueli, A. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods* (2008), FM '08, pp. 35–51.

[69] Zhang, L., Yang, G., Rungta, N., Person, S., and Khurshid, S. Invariant discovery guided by symbolic execution. In *The Java PathFinder Workshop* (2013).

[70] Zhang, Z., and Koutsoukos, X. Generic value-set analysis on low-level code. In *Cyber-Physical Systems Workshop* (2014).

[71] Zuck, L., Pnueli, A., Fang, Y., and Goldberg, B. Voc: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science 9*, 3 (2003), 223–247.