Sound Loop Superoptimization for Google Native Client

Berkeley Churchill
Stanford University
berkeley@cs.stanford.edu

Rahul Sharma *
Microsoft Research India
rahsha@microsoft.com

JF Bastien
Stanford University
paper@jfbastien.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

Software fault isolation (SFI) is an important technique for the construction of secure operating systems, web browsers, and other extensible software. We demonstrate that superoptimization can dramatically improve the performance of Google Native Client, a SFI system that ships inside the Google Chrome Browser. Key to our results are new techniques for superoptimization of loops: we propose a new architecture for superoptimization tools that incorporates both a fully sound verification technique to ensure correctness and a bounded verification technique to guide the search to optimized code. In our evaluation we optimize 13 `libc` string functions, formally verify the correctness of the optimizations and report a median and average speedup of 25% over the libraries shipped by Google.

1. Introduction

Software fault isolation (SFI) is a sandboxing technique to isolate untrusted code from a larger system [8, 28, 38, 44, 46]. Google Native Client (`NaCl`), a SFI system shipped with Google Chrome, safely allows untrusted extensions to be loaded into the web browser [46]. `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 [38]. 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 [26, 30].

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). This paper 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`

* work done at Stanford University

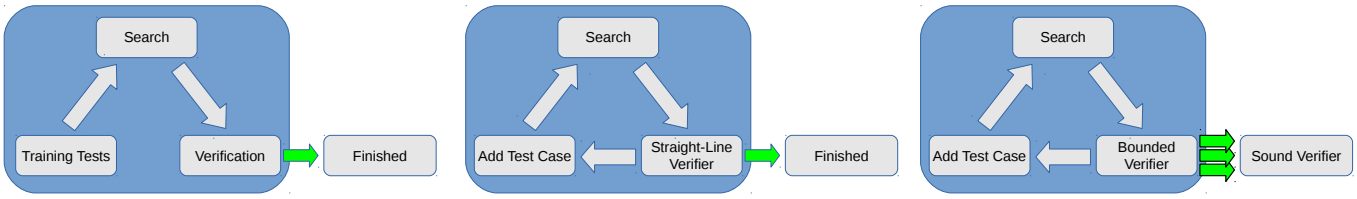
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '17 April 8–12, 2017, Xi'an, China

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: 3037697.3037754



(a) Search without automatic test case generation [39, 40].

(b) Search with automatic test case generation for straight-line code [36].

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

Figure 1: Search architecture of different stochastic superoptimization tools.

code. While prior work [39] 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 [36, 39] for our initial `NaCl` superoptimizer. However, there are three 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 1b). However, there is no general method for generating counterexamples from sound verification 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 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: the fastest discovered program is usually the last one found. 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 both pass all tests. Retaining only the fastest such rewrite usually results in failure during verification.

Our solutions to these first 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 1c. A bounded verifier performs a partial proof of equivalence. For a user-supplied bound parameter k, it checks whether the target t and the rewrite r 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 [4, 9].

The third problem is the exponential growth in the number of aliasing cases that may be required to reason about memory. The number of ways the memory accesses in two programs can overlap grows rapidly with the number 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. In our case, our tool does not have access to source code information and writing new abstract semantics for each x86-64 instruction

is prohibitively time-consuming. We instead introduce *alias relationship mining*, a new technique which uses test data to soundly reduce the number of aliasing cases by several orders of magnitude. Alias relationship mining allows our verifiers to handle more difficult examples reliably without timing out. Our initial implementation based on a standard *flat memory model* [45] was much less predictable.

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, they are ubiquitous in assembly code; any real 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. However, none of our techniques are specialized to string functions; our tool does not depend on the data structures a program uses.

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 increases the number of verification tasks that can be completed. In summary, this paper 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 describe *alias relationship mining*, a novel technique to use data from test cases to improve the performance of a bounded verifier that handles potentially aliased memory locations soundly.
- We detail enhancements to DDEC, the sound verification algorithm for proving loop equivalence introduced in [39]. 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 paper is organized as follows. Section 2 illustrates the operation of the bounded verifier on a `NaCl` code example. Section 3 details the implementation of the bounded and sound verification techniques. Section 4 discusses the extensions to STOKE required for generating `NaCl` code. In Section 5, we demonstrate our contributions empirically. We conclude with a summary of related work in Section 6 and closing remarks in Section 7.

2. Motivating Example

Figure 2 is an example of a target and an incorrect rewrite we use to demonstrate the utility of bounded verification in search. Adapted from the `wcpcpy` `libc` routine (string copy for wide character strings), the target is equivalent to the following C code. Note that, even though the platform is 64-bit, `NaCl` treats all pointers as 32-bit. This example uses 32-bit wide characters.

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

The target and rewrite code in Figure 2 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. 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 $k_1 + \%\texttt{r15} + k_2X$. 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.

```
    # Target                  # Rewrite

1   .begin:                   movl esi, esi
2    movl esi, esi            movl (r15,rsi), edx
3    movl (r15,rsi), edx      addl 4, esi
4    movl edi, eax            nop (23)
5    addl 4, esi             .begin:
6    movl edi, edi            movl edi, eax
7    movl edx, (r15,rdi)      movl edi, edi
8    addl 4, edi              movl edx, (r15,rdi)
9    testl edx, edx           shrl 1, edx
10   jne .begin               je .exit
11   retq                     movl esi, esi
12                            movl (r15,rsi), edx
13                            addl 4, esi
14                            jmpq .begin
15                            nop (31)
16                           .exit:
17                            retq
```

Figure 2: A target and rewrite for `wcpcpy`. The code is ATT syntax with `%` and `$` prefixes removed for space. This benchmark performs a string copy of 4-byte wide characters.

- 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.

2.1 Bounded Verifier

The *bounded verifier* works as follows. For a given bound k, we enumerate the set of all possible paths through the target t and the rewrite r 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 t executes p and r 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 (Section 3.2). 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 [13] to check if $\exists x. g_p(x) \wedge g_q(x) \wedge f_p(x) \neq f_q(x)$. If such an x exists, then we have generated a counterexample which can be used as a new test case for the search. Otherwise, t and r 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.

2.2 Alias Relationship Mining

Consider the sub-task of proving the equivalence of paths p_2 and q_2, which corresponds to performing a string copy on 4-byte wide character strings where the input has only one non-null character followed by a null terminating character. We explicitly unroll the code along these paths as shown in Figure 3. When these paths are taken, there are eight memory accesses in total, four in the target (labeled a, b, c, d) and four in the rewrite (labeled a', b', c', d'). In this example, the accesses are in one-to-one correspondence; when executed, the addresses accessed by the target are always the same as those accessed by the rewrite. 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 `%rsi`$-$ `%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

```
# Target                  # Rewrite

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: A target and rewrite for `wcpcpy` unrolled 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.

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 over p_2 and q_2, we must run a total of 16 queries to the SMT solver, one for each possible aliasing configuration.

Past work on STOKE models 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 [39] took two hours to validate two assembly sequences with less than 10 LOC and two dereferences each. We introduce a technique called *alias relationship mining* (ARM) which constructs a minimal set of aliasing configurations required to perform the proof.

In ARM we run the unrolled target and rewrite 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 for paths p, q 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 bounded validator a total of 16 times, one for each possible aliasing configuration. For details, see Section 3.2.

3. Implementation

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

3.1 Bounded Verifier

The *bounded verifier* takes a target t and a rewrite r 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 $p \in \mathit{Path}_T, q \in \mathit{Path}_R$, 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 as a set of pseudo-registers as described in Section 3.2. 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 [23]. Suppose path p executes instructions $s_1^t, \ldots, s_m^t$ through the target and path q executes instructions $s_1^r, \ldots, s_n^r$ through the rewrite. Let x_0 and y_0 denote the start states of p and q. 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.,

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

C_3 represents the execution of the rewrite through q:

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

C_4 encodes path conditions; if s_i^t 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 r 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.

3.2 Alias Relationship Mining

The unique contribution in our design of the bounded verifier is an efficient and sound approach to prove equivalence in the presence of possibly aliased memory locations. Given fixed-length paths p and q through the target and rewrite our goal is to model every memory access as a read or write to a set of pseudo-registers. The first step is to enumerate statically all the memory accesses, as done in Figure 3 of Section 2.2. On a single execution of each path, we know that each memory dereference will only be run once. 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. Let $A(\mu)$ denote the symbolic representation of the address dereferenced by access μ; we derive $A(\mu)$ through symbolic execution of p and q. 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 p or q 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 always valid. In the absence of test cases, an alternate technique 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.

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 bounded validator once for every such aliasing configuration.

Alias relationship mining is well suited to data-driven systems such as STOKE where learning relationships from test data is easy. There is a great deal of existing work in the symbolic execution, program analysis and bounded model checking communities on memory models (see Section 6), and there are other approaches to avoiding worst-case behavior in analyzing aliasing. However, our approach is advantageous for our application for several reasons. First, we had no need to write a new set of per-instruction abstract semantics to perform a separate pointer analysis; this task is a prohibitively time consuming for the x86-64 instruction set. Second, we accurately handle cases where pointers alias. Most importantly, it scales better than the flat memory model that we implemented with Z3; see Section 5.

3.3 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 t is *equivalent* to r if, when we run t and r 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) [39], 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 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 [39] this would cause DDEC to fail. In this work, we have added fixedpoint iterations to eliminate spurious invariants.

3.3.1 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.

3.3.2 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) take 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 $S_\lambda^j = \{(x_i, y_i) \in S_\lambda : C_\lambda^j(x_i, y_i)\}$. 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: (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.

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

Figure 4: 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.

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 [14] 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 in invariant asserting the target and rewrite have identical heap states.

3.3.3 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 [18]. 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. 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.

4.1 Transformations

Optimizing `NaCl` code requires more aggressive transformations compared to the ones described in previous works that use STOKE [36, 39, 40]. 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 [39]. 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 nop moves* insert an extra nop into the program $(\star)$.
6. *Replace nop moves* replace an instruction with a string of nops whose binary representation has the same length of 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.

4.2 Cost Function

A *cost function* produces a score for each rewrite, where lower values are better. The cost function guides the search towards desirable rewrites. As described earlier, well-formed `NaCl` code must obey certain rules, such as those on instruction alignment and memory accesses.

The `NaCl` cost function assigns a score 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 [36]: 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.

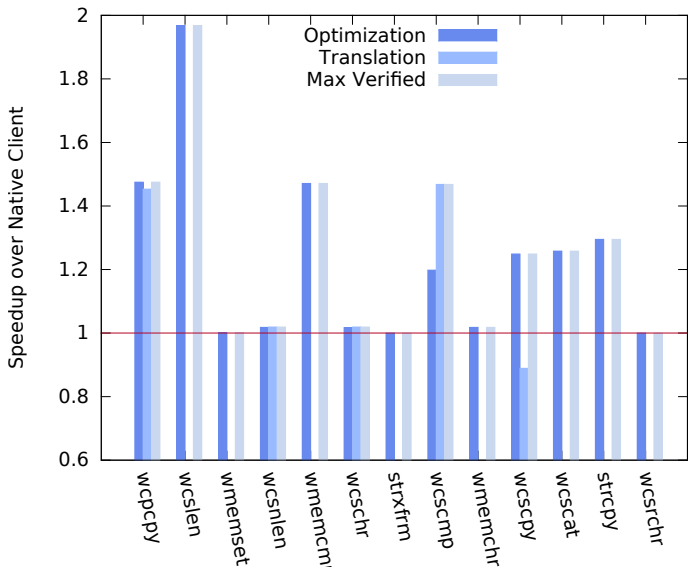


Figure 5: Speedups by benchmark. For each benchmark, the speedup over the original `NaCl` library is shown. The bars correspond to the optimization experiment, the translation experiment, and the best rewrite we verified. The 'optimization mode' much more reliably produces a verifiable result, but 'translation mode' sometimes offers significant improvements.

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 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.

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 implementation with our new system that uses the bounded verifier. Finally, we compare the performance of the alias relationship mining to the flat memory model.

5.1 Experiment Setup

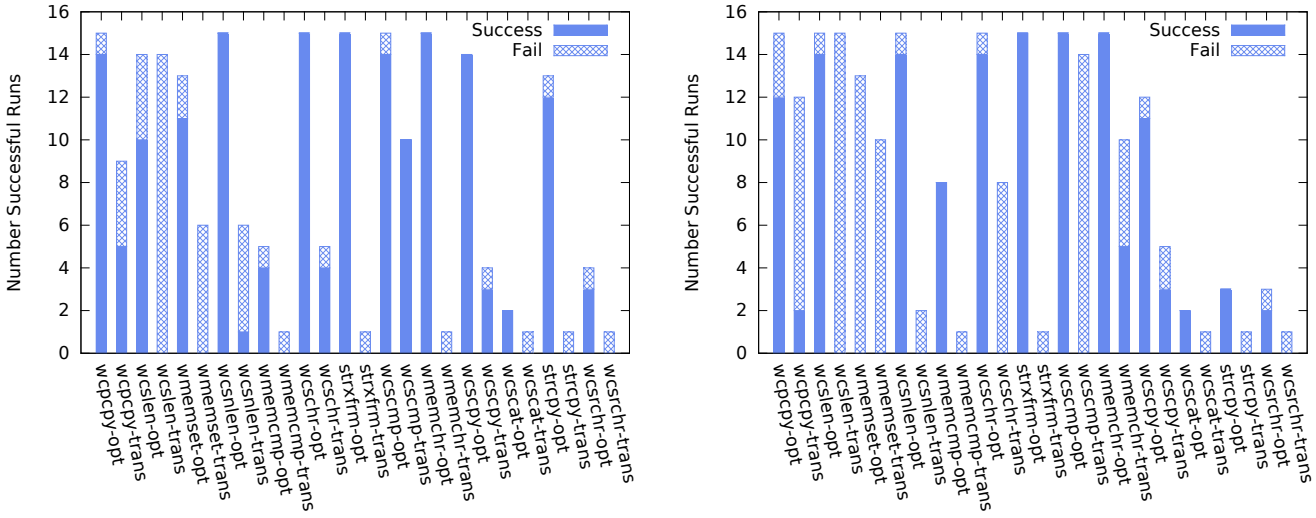
Our goal is to improve the performance of each of the 13 `libc` string functions and prove correctness of the optimized code. For each benchmark we perform two experiments, optimization and translation. In *optimization mode*, we initialize the rewrite with the code shipped with `NaCl` and run STOKE 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 is naive and 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 1.

5.1.1 Performance Results

The performance results are shown in Figure 5 and Table 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



(a) Alias Relationship Mining

(b) Flat Memory Model

Figure 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 alias relationship mining 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.

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 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 for 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` compiler is overly conservative: it aligns every jump to a 32-byte boundary instead of only indirect jumps. Table 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.

5.1.2 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 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. In no cases did the verification fail for a correct rewrite, meaning our choice of cutpoints and loop invariants were sufficient.

One observes that the 180 cases where sound verification failed due to an incorrect candidate rewrite contradicts the often-assumed "small scope hypothesis" [2, 24, 31]. 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.

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 alias relationship mining (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 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

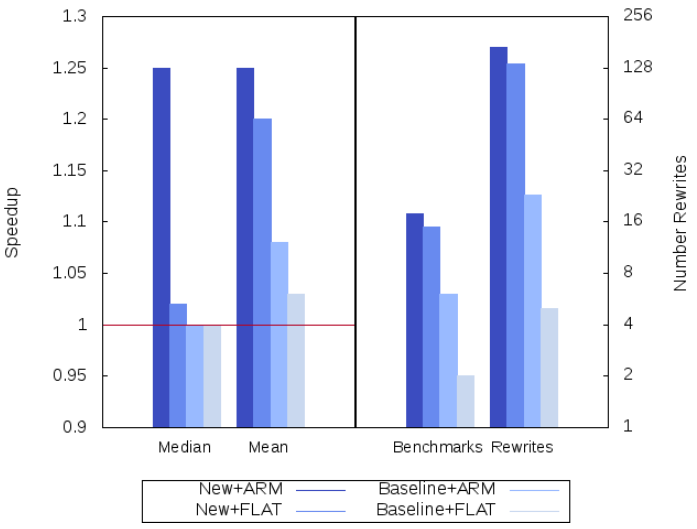


Figure 7: Comparison of implementations. For each of four implementations, the median and mean best speedups are plotted across the 13 benchmarks in translation and optimization modes. The number of benchmarks where an improved result was found (maximum is 26 considering optimization and translation 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 alias relationship mining also improves the number and quality of verified rewrites.

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.

5.3 Memory Model Performance

The key difference between alias relationship mining and the flat memory model is that ARM reasons about the aliasing of memory locations outside the SMT solver, while the flat memory model offloads this work to Z3. Consequently, the performance of the flat model is subject to the peculiarities of the implementation of the underlying solver. We find that the flat model outperforms ARM in many small examples, but doesn't scale predictably. ARM scales more gracefully and can handle a larger proportion of the verification tasks.

Figure 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.

As a separate benchmark, we took 1128 DDEC verification tasks derived from search outputs 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 better average time of 24s per task compared to ARM with average time 554s per task.

6. Related Work

The techniques used in our work are related to several lines of research, including symbolic execution, bounded model checking, translation validation and binary analysis. Our work is most related to the area of stochastic program search.

Superoptimization. This paper most directly advances work on superoptimization. The original superoptimization paper by Massalin and even some recent works check for correctness using test cases alone [29, 37]. Other authors have used formal verification techniques for straight-line code [21, 25, 36]. More recently, DDEC made verification of loops in a superoptimization setting possible, but did not generate new test cases to guide the search and used an inefficient memory model; thus, it couldn't scale beyond programs ten lines long [39]. LENS [27] formulates stochastic search for straight-line code over graphs and offers improvements over [36]. This work is the first work that allows for end-to-end verification of superoptimized loops in a real-world setting.

Equivalence Checking is an old problem with an extensive literature. Several works develop equivalence checking algorithms for low level code, but they do not handle unbounded loops, and often use a flat memory model [3, 12, 16, 17, 41]. Regression verification [19] compares two successive versions of a program and has been applied to integer programs [15]. Abstract Semantic Differencing [32] proves (partial) equivalence of loops. However, it uses numerical domains that are inapplicable to the bitvector arithmetic in x86-64 assembly.

Binary Analysis. Many authors have developed tools for static analysis of x86-64 code, including symbolic execution tools [7, 10, 42]. Our bounded verifier is similar to several of these tools. Rudder is a symbolic execution tool which used to find security bugs in x86-64 binaries. They also support two memory models, a flat model [6], and a model that proves bounds on addresses a memory access can dereference [43]. Both are sound with respect to aliasing. The latter model is similar to alias relationship mining but yields less precise information. CODESURFER/X86 is a static analysis tool for x86 that uses *value set analysis* to derive memory aliasing information. VSA gives precise results for strided intervals; however, it requires implementing value set semantics for each instruction [5]. SYMDIFF has been used to verify the equivalence of x86-64 code generated by different compilers [22], but makes unsound assumptions. In particular, the treatment of aliasing is unsound and the register values are treated as unbounded mathematical integers rather than as bitvectors.

Translation Validation. Translation validation [20, 30, 34] verifies the equivalence of a program before and after a compiler's optimization pass. Necula's work [30] is akin to our sound verification in that it infers and checks a simulation relation between the two programs. A difference is that Necula constructs the simulation relation through static analysis and information provided by the compiler, whereas we derive the simulation relation from data.

Bug Finding and Symbolic Execution. There are several bug finding tools that cross check implementations and report differences. These tools bound the number of loop iterations and are similar in function to our bounded verifier. For example, UC-KLEE [35] tests equivalence of code using symbolic execution, but handles cases where pointers alias unsoundly. Differential symbolic execution (DSE) and Currie [11, 33] handle pointer aliasing soundly, but model common parts of programs with uninterpreted functions to reduce the complexity of the constraints. This abstraction isn't suited for STOKE because it may result in false-positives, and the target and rewrite may not have enough similarity to gain a benefit.

7. Conclusion

In this paper we extend stochastic superoptimization to the domain of software fault isolation. Our key contribution is a new technique for superoptimization of loop kernels, featuring both a bounded and a sound validator, and alias relationship mining, a novel technique to improve the scalability of verification in the presence of aliased memory. We demonstrate that these advances produce efficient and formally verified `NaCl` code.

Acknowledgments

The authors would like to acknowledge Stefan Heule for his work and assistance on the STOKE system. This work was supported by NSF grants CCF-1160904 and CCF-1409813, a Microsoft Fellowship, and by DARPA under agreement number FA84750-14-2-0006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied of DARPA or the U.S. Government.

References

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

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

[3] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *Computer Aided Verification (CAV)*, 2005.

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

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

[6] D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, School of Computer Science, Carnegie Mellon University, 2007.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification (CAV)*, 2011.

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

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

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

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

[12] D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, 2000.

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

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

[15] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Automated Software Engineering (ASE)*, 2014.

[16] X. Feng and A. J. Hu. Automatic formal verification for scheduled VLIW code. In *Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES-SCOPES)*, 2002.

[17] X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *Embedded Software (EMSOFT)*, 2005.

[18] C. Flanagan and K. R. M. Leino. Houdini: An annotation assistant for ESC/Java. In *Formal Methods Europe (FME)*, 2001.

[19] B. Godlin and O. Strichman. Regression verification. In *Design Automation Conference (DAC)*, 2009.

[20] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1), 2005.

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

[22] C. Hawblitzel, S. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? Static cross-version compiler validation. In *Foundations of Software Engineering (FSE)*, 2013.

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

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

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

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

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

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

[29] H. Massalin. Superoptimizer - a look at the smallest program. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1987.

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

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

[32] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *Static Analysis Symposium (SAS)*, 2013.

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

[34] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1998.

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

[36] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[37] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

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

[39] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, 2013.

[40] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Conditionally correct superoptimization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[41] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction*, 2005.

[42] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[43] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Information Systems Security (ICISS)*, 2008.

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

[45] W. Wang. *Partitioned Memory Models for Program Analysis*. Ph.D., New York University, 2016.

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