一种静态分析的方法

Testing Static Analyses for Precision and Soundness

Jubi Taneja
University of Utah
USA
jubi@cs.utah.edu

Zhengyang Liu
University of Utah
USA
liuz@cs.utah.edu

John Regehr
University of Utah
USA
regehr@cs.utah.edu

Abstract

背景：编译器中静态分析的意义

Static analyses compute properties of programs that are true in all executions, and compilers use these properties to justify optimizations such as dead code elimination. Each static analysis in a compiler should be as precise as possible while remaining sound and being sufficiently fast. Unsound static analyses typically lead to miscompilations, whereas imprecisions typically lead to missed optimizations. Neither kind of bug is easy to track down.

目的 Our research uses formal methods to help compiler developers create better static analyses. Our contribution is the design and evaluation of several algorithms for computing sound and maximally precise static analysis results using an SMT solver. These methods are too slow to use at compile time, but they can be used offline to find soundness and precision errors in a production compiler such as LLVM. We found no new soundness bugs in LLVM, but we can discover previously-fixed soundness errors that we re-introduced into the code base. We identified many imprecisions in LLVM's static analyses, some of which have been fixed as a result of our work.

贡献

缺点

贡献

CCS Concepts • **Software and its engineering → Compilers**; **Formal software verification**; **Automated static analysis**; • **Theory of computation** → *Design and analysis of algorithms.*

Keywords Dataflow analysis, Soundness, Precision

ACM Reference Format:
Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377927

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377927

1 Introduction

Static analysis is one of the basic technologies that enables compilers to generate optimized executables for code in high-level languages. For example, an integer overflow check can be removed when the operation provably does not overflow, and an array bounds check can be removed when the access is provably in-bounds. In each case, the necessary proof is implied by the results of a static analysis.

举例说明静态分析对于编译优化的重要性

Production-quality optimizing compilers have typically had significant engineering effort put into their static analyses. For example, LLVM currently has about 80,000 lines of C++ in its lib/Analysis directory, which contains much, but not all, of its static analysis code. GCC does not have an analogous directory structure that makes it easy to find static analyses, but for example its integer range analysis alone (tree-vrp.c) is 6,929 lines of C. This code is tricky to get right and, so far, developers are implementing static analyses without any help from formal-methods-based tools.

这段主要讲在工业界LLVM和GCC上静态分析的源码都是复杂且长难以理解的；开发人员没有正式的方法来检测静态分析

We have developed algorithms that use an SMT solver to compute sound and maximally precise results for several important dataflow analyses used by LLVM:

介绍了本文的工作

- Known bits: a forward analysis attempting to prove that individual bits of a value are always either zero or one
- Demanded bits: a backward analysis attempting to prove that individual bits of a value are "not demanded": their value is irrelevant
- Integer ranges: a forward analysis attempting to prove that an integer-typed value lies within a sub-range, such as [5..10]
- A collection of forward analyses determining Boolean properties of a value, such as whether it is provably non-zero or a power of two

Since 2010, 55 soundness bugs in LLVM's forward bit-level analyses have been fixed. Similarly, three soundness bugs have been fixed in its demanded bits analysis, and 23 soundness errors have been fixed in its integer range analysis. On the other side, static analyses that are insufficiently precise can impede optimization; in one case, increasing the precision of a static analysis in LLVM improved the performance of a Python benchmark by 4.6% [14]. The goal of our work is to develop formal-methods-based algorithms that can be used to find both imprecisions and unsoundnesses in static analyses implemented in production-quality compilers.

目标

方法大致介绍

Our algorithms work on concrete code fragments, which we gather by compiling real applications. For each such fragment, we compute a sound and maximally precise dataflow fact and then compare this against dataflow information computed by LLVM. The interesting outcomes are where LLVM's result is "more precise" than ours (indicating a soundness error in LLVM) or LLVM's result is less precise than ours (indicating an imprecision in LLVM that may be worth trying to fix). We have reported several precision errors to the LLVM developers, and some of these have been fixed. We did not find any soundness errors in the latest version of LLVM, but we have verified that our method can find previously-fixed bugs. Finally, we have created a version of LLVM that uses our dataflow results instead of its own. This compiler is very slow, but it demonstrates the value of increased precision by sometimes creating binaries that are faster than those produced by the default version of LLVM.

结果：速度较慢；只找到了精度错误但是稳健性错误没有发现新的

2 Background

Our work builds on existing results in dataflow analysis, and our implementation builds on the Souper superoptimizer [22].

2.1 Dataflow Analysis

Dataflow analysis is a body of techniques for computing facts that are true in all executions of the program being analyzed. A dataflow analysis should remain sound, but is allowed to be imprecise, for example by over-approximating the set of memory locations that a pointer might refer to. Approximation is necessary in order to sidestep decidability problems.

Programs are made of concrete operations such as addition that operate on concrete values. Dataflow analyses employ abstract operations that operate on abstract values; each abstract value represents a set of concrete values. For example, the integer range [6..10] represents the set $\{6, 7, 8, 9, 10\}$. A *concretization function* γ maps an abstract value to the set of concrete values it represents, and an *abstraction function* α maps any set of concrete values to the most precise abstract value whose concretization is a superset of the given set. For any abstract value e, it should always be the case that $\alpha(\gamma(e)) = e$. However, it is not the case that, for an arbitrary set of concrete values S, $\gamma(\alpha(S)) = S$. For example, in the abstract domain of integer ranges, $\gamma(\alpha(\{5, 8\})) = \{5, 6, 7, 8\}$. Abstract values form a lattice or semilattice, with the lattice order corresponding to the subset relation among the concretization sets of the values in the lattice.

Let $\hat{f}$ be an abstract operation, such as addition over integer ranges. If $\hat{f}([6..10], [1..2]) = [7..12]$, then in this case $\hat{f}$ is returning the most precise possible result. It is always possible to create a sound and maximally precise abstract operation by concretizing its arguments, applying the corresponding concrete operation to the cross product of the concretization sets, and then applying the abstraction function to the set containing the resulting concrete values. Obviously this implementation is too slow to use in practice unless concretization sets are small. The challenge in creating good abstract operations is to make them sound and as precise as possible, while making them fast enough so that static analyses built upon them do not slow down the overall compilation too much. Addition for integer ranges is an easy case: the lower bound of the output interval is just the sum of the lower bounds of the input intervals, etc. Other cases are much more difficult.

2.2 Static Analysis in LLVM

LLVM's intermediate representation (IR) is a good substrate for many static analyses. First, rules that are tricky and implicit at the programming-language level, such as type conversions and ordering of side-effects in expressions, are made explicit. Second, the SSA [9] form makes it easy and efficient (in the absence of pointers, at least) to follow the flow of data through a function. Third, since LLVM contains competent implementations of a variety of IR-level optimizations such as constant propagation, dead code elimination, function inlining, arithmetic simplifications, and global value numbering, many small obstacles to static analysis are optimized away before they need to be dealt with.

Known Bits This analysis determines which bits of a value are zero or one in all executions. Known bits are used in the preconditions of peephole optimizations. For example, a signed division-by-constant can be implemented more efficiently if the dividend provably has a zero in its high-order bit. These precondition checks are ubiquitous in LLVM's instruction combiner, a 35,000 line collection of peephole-like optimizations.

Let f be a function composed of LLVM instructions, W be the width in bits of f's output, and $\mathbb{K}$ be the result of the known bits analysis for the output of f. Let $f(x)_i$ and $\mathbb{K}_i$ represent the values of f and the analysis result at bit position i. Each bit of the analysis result is known to be zero, known to be one, or else unknown. The known bits analysis is sound if:

$$\forall i \in 0 \ldots W-1,\ \forall x : \mathbb{K}_i = 0 \implies f(x)_i = 0 \wedge \mathbb{K}_i = 1 \implies f(x)_i = 1$$

The known bits analysis is maximally precise if it always concludes that a bit is known when it is sound to do so:

$$\forall i \in 0 \ldots W-1,\ (\forall x : f(x)_i = 0) \implies \mathbb{K}_i = 0 \wedge (\forall x : f(x)_i = 1) \implies \mathbb{K}_i = 1$$

Number of Sign Bits The number of sign bits is the number of high-order bits that provably all hold the same value. Every value trivially has at least one sign bit, and values can accrue additional sign bits by, for example, being sign extended or arithmetically right shifted. The number of sign

bits analysis can be used to reduce the number of bits allocated to a variable. This analysis's criteria for soundness and maximal precision are analogous to those for known bits, so we omit them.

Single-bit Analyses LLVM provides a number of analyses that provide a single bit of information; these are:

- Value is provably non-zero
- Value is provably negative
- Value is provably non-negative
- Value is provably a power of two

Their criteria for soundness and maximal precision are analogous to those for known bits.

Integer Range Analysis Integer ranges are used to optimize away comparisons, for example $[0, 100) < [200, 205)$ can be simplified to "true." LLVM's Correlated Value Propagation pass attempts to optimize every comparison in a program in this fashion.

Lazy Value Info (LVI) is LLVM's version of the classic integer range analysis; it computes a *constant range* that has one of four forms:

- Empty set: concretization set is empty
- Full set: concretization set is all values of the integer type
- Regular range $[a, b)$ with $a <_u b$: concretization set contains all values $\geq a$ and $< b$
- Wrapped range: $[a, b)$ with $a >_u b$: concretization set contains all values either $\geq a$ or $< b$

Here the $>_u$ and $<_u$ operators indicate unsigned integer comparison. So, for example, the concretization set of the wrapped range $[11, 10)$ would include every value except 10. Constant ranges where $a = b$ are invalid unless $a = b = 0$ or $a = b =$ UINT_MAX, which respectively represent the empty and full sets.

This analysis is sound if the concretization set of the analysis result $\mathbb{R}$ is a superset of the union of the results of applying the concrete operation to all possible inputs:

$$\gamma(\mathbb{R}) \supseteq \bigcup_{\forall a} f(a)$$

Unfortunately, the corresponding comparison in the abstract domain—ensuring that the analysis result is high enough in the lattice—does not work because this abstract domain does not actually form a lattice. To see this, notice that the abstraction function is not forced to return a unique best result, but rather must decide between a regular and wrapped constant range. We sidestep this problem by dictating that an integer range is maximally precise if it always returns a result whose concretization set is as small as possible. Gange et al. [13] presented a nuanced discussion of using non-lattice abstract domains for static analysis.

Demanded Bits So far, the analyses from LLVM that we have considered are *forward analyses*: they track data flow facts through the program in the same direction that data flows during execution. The *demanded bits* analysis is a *backwards program analysis* that pushes facts in the opposite direction; it looks for bits whose value does not matter. For example, if a 32-bit value is truncated to 8 bits (and has no other uses in the program), then the demanded bits analysis can return false for its top 24 bits, indicating that they are not demanded.

A demanded bits analysis for f is sound if every not-demanded bit of the input can be set to either zero or one without changing the result of the computation:

$$\forall i \in 0 \ldots W-1, \mathbb{D}_i = 0 \implies$$
$$\forall x : (f(x) = f(\text{setBit}_i(x))) \wedge (f(x) = f(\text{clearBit}_i(x)))$$

Here $\mathbb{D}$ is the result of the demanded bits analysis and $\text{setBit}_i()$ and $\text{clearBit}_i()$ are functions that respectively force bit i of their inputs to one and zero.

This analysis is maximally precise if, for every demanded bit, there exists an input value causing f to return a different result than it produces when that bit of input is forced to either zero or one:

$$\forall i \in 0 \ldots W-1, \mathbb{D}_i = 1 \implies$$
$$\exists x : (f(x) \neq f(\text{setBit}_i(x))) \vee (f(x) \neq f(\text{clearBit}_i(x)))$$

When some bits are not demanded, it is sometimes the case that a computation can be replaced by a simpler one that only produces the desired result for the demanded bits. When no bits are demanded, a value is dead and the instructions producing it can be removed (assuming their values have no other uses).

2.3 Reasoning about LLVM using Souper

Souper [22] is an open-source superoptimizer: it runs as an LLVM middle-end optimization pass, extracting LLVM IR into its own IR. In its default mode of use, it employs synthesis to compute more efficient versions of the extracted code and—when synthesis succeeds—applies the discovered optimizations to the code being compiled.

For this work, we reused some parts of Souper as infrastructure for computing precise dataflow facts. To do this, the main thing we needed from Souper was its ability to extract LLVM IR into an internal representation that we can use to build customized queries for SMT solvers, in order to implement the algorithms described in the next section. Each extracted piece of code, which we call a "Souper expression," represents an arbitrary-sized directed acyclic graph of LLVM instructions. These can come from multiple basic blocks, taking phi nodes into consideration, and adding path conditions corresponding to branches in the LLVM code. Souper's main limitations are that it cannot see through memory references, function calls, or multiple loop iterations. Most of the other parts of Souper remain unused by our work, with the exception of its ability to synthesize an integer constant

meeting a specification—we use that as part of the integer range computation described in Section 3.3.3.

3 Testing Dataflow Analyses for Soundness and Precision

This section describes our work, which must solve several sub-problems:

- Finding representative LLVM IR on which to test the compiler's dataflow analyses (Section 3.1)
- Ensuring that LLVM's static analyses and our algorithms for computing dataflow results both see the same code, so that their results are directly comparable (Section 3.2)
- Computing precise dataflow results using an SMT solver (Section 3.3)

The high-level structure of our method is shown in Figure 1.

3.1 Finding Test Inputs

测试输入的来源

We require concrete code fragments to use as bases for comparison between LLVM's dataflow implementations and ours. We got these by compiling all of the C and C++ benchmarks in SPEC CPU 2017[1] with Souper plugged into LLVM as a middle-end optimization pass, and with Souper's synthesis disabled. This has the effect of loading a large number of Souper expressions into a cache that we subsequently used as a source of test inputs.

After compiling SPEC CPU 2017, we ended up with 269,113 unique Souper expressions. 71.6% of these were encountered more than once during compilation, 11.4% more than 10 times, and 1.6% more than 100 times.

3.2 Enabling Comparable Results

A problem we had to solve is that we cannot easily control what LLVM is willing to do to get precise dataflow results, making it hard to put LLVM's analyses and ours on an even playing field. As an example, LLVM's integer range analysis is sometimes interprocedural, because some passes will mark function calls with "range metadata" gathered from callees. This kind of feature puts us in the awkward position of potentially having to either modify LLVM or else emulate all of LLVM's precision-increasing mechanisms in order to get comparable results. We developed a different solution: instead of running LLVM's static analyses on its original IR, we convert Souper's IR back into LLVM IR and then analyze that. This is accomplished using the `souper2llvm` tool shown in Figure 1. This strategy ensures that LLVM's analyses and ours are computing dataflow facts over exactly the same code.

[1] https://www.spec.org/cpu2017/

3.3 Algorithms for Maximally Precise Dataflow Analyses 最大精确的数据流分析的算法

Like other synthesis problems, computing precise dataflow results using an SMT solver is a search problem that in practice requires multiple solver calls to arrive at the best answer. Our algorithms achieve maximal precision and perform reasonably well by exploiting the structure of the abstract domains that we target. In some cases these algorithms are trivial. For example, all of the single-bit analyses described in Section 2 can be computed in a maximally precise fashion using a single solver call which simply checks whether the property implied by the fact holds or not. Another analysis, the one computing the number of sign bits, is nearly as easy, since there are only $n-1$ non-trivial possibilities for an n-bit value. We simply try all alternatives, one after the other, starting with the most precise result (n sign bits). In contrast, an exhaustive search strategy is intractable for:

- demanded bits: 2^n alternatives
- known bits: 3^n alternatives
- integer ranges: $4^n - 2^n + 2$ alternatives

The rest of this section is about how to compute these dataflow facts more efficiently and without giving up maximal precision. Our arguments for maximal precision are all contingent on the SMT solver returning a definite answer (SAT or UNSAT) for every query, as opposed to timing out. It is easy to construct an adversarial example (involving 64-bit divisions, for example) that cannot be analyzed in practice, using our methods, because it results in SMT queries that cannot be solved in a reasonable amount of time by a state-of-the-art solver.

Algorithm 1 maximally precise known bits

```
1: procedure computeKnownBits(F)
2:     for i ← 0 . . . Width(F) − 1 do
3:         if askSolver(F = F.clearBit(i)) then
4:             Result_i ← 0
5:         else if askSolver(F = F.setBit(i)) then
6:             Result_i ← 1
7:         else
8:             Result_i ← Unknown
9:     return Result
```

3.3.1 Known Bits

Algorithm 1 computes known bits efficiently and maximally precisely. The algorithm is not difficult: it simply guesses that each bit is always zero and then always one. In this pseudocode (and also in the pseudocode for the next two algorithms) we elide error-checking code; it should be assumed that the algorithm terminates with a sound (but likely imprecise) result whenever the solver runs out of time or memory. The input, F, in this algorithm is a function computed by some Souper expression. So then, "getInputs(F)"

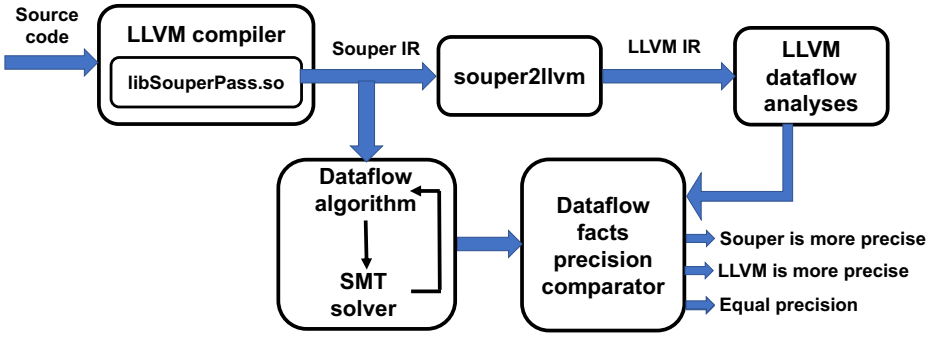


Figure 1. Overview of our method for finding imprecisions or unsoundnesses in LLVM's dataflow analyses. libSouperPass.so is dynamically loaded into LLVM to allow Souper to run as an optimization pass. souper2llvm is a utility we created that translates Souper IR into LLVM IR.

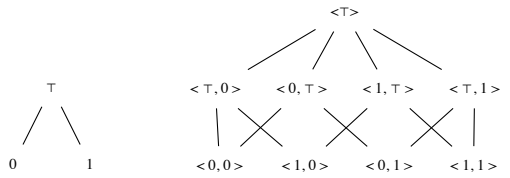
returns the inputs to the Souper expression F, "F.setBit(input, i)" pins bit 1 of one of the inputs to F, etc.

It is a little trickier to prove that bit-by-bit search, which requires only $2n$ solver queries, is maximally precise. The argument is based on *separability*: a property held by a dataflow analysis if the aggregate dataflow function can be viewed as a product of simpler functions on individual data items [15]. If known bits is separable at the bit level, then our bit-by-bit algorithm must always return the most precise result. The abstract domain for known bits forms a *join semilattice* [23]; every subset of its elements has a join, but not a meet. (Throughout this discussion, we'll refer to join semilattices as lattices when this does not seem to risk ambiguity.) The lattice for one bit contains three abstract values as shown in Figure 2(a), where elements 0 and 1 are more precise than the top element $\top$ and the join of elements 0 and 1 is $\top$:

$$0 \sqsubseteq \top, \quad 1 \sqsubseteq \top, \quad 0 \sqcup 1 = \top$$

The lattice for two bits, as shown in Figure 2(b), is created by taking a cross-product of two one-bit lattices. The elements of the cross-product lattice, $L_1 \times L_2$, are tuples of the form: $<X_1, X_2>$ such that every $X_i \in L_i$ lattice.

To prove separability of known bits analysis, we need to prove the following three properties as defined by Khedker et al. [15].



(a) For one bit **(b)** For two bits

Figure 2. The known bits abstract domain

Property 3.3.1. For all elements in a lattice, the dataflow property is independent, and thus can be ordered element-wise.

$\forall X, Y \in L: X \sqsubseteq Y \equiv <X_1 \sqsubseteq Y_1, X_2 \sqsubseteq Y_2, \ldots, X_n \sqsubseteq Y_n>$

Property 3.3.2. For each element X in a cross-product lattice, where X is a tuple of component elements, there exists a set of functions f in a bigger set of transfer functions $\mathfrak{F}$ for a dataflow framework, that when applied over aggregate element X is equivalent to a tuple of smaller abstract functions that are applied on component elements of X. In short, the separable function space can be factored into a product of the functions' spaces for individual data items.

$\forall f_i: L_i \rightarrow L_i, 1 \leq i \leq n$, such that $\forall f \in \mathfrak{F}, f(X) \equiv <f_1(X_1), f_2(X_2), \ldots, f_n(X_n)>$.

Both of these properties are clearly held by a lattice like known bits that is created by concatenating items corresponding to individual bits: there is no interaction across bits in this abstract domain.

Property 3.3.3. The height of each L_i is bounded by a constant.

The height of the known bits lattice is one larger than the number of bits being analyzed.

Separability establishes that a transfer function applied to an element of a cross-product lattice gives the same solution as the single-bit transfer functions applied individually. This indicates that both approaches compute the same fixed point. Thus, our bit-by-bit algorithm is maximally precise. (Our reasoning here is empirically supported by the fact that over a large number of expressions harvested from LLVM IR, our known bits result was never less precise than the one computed by LLVM.)

3.3.2 Demanded Bits

Algorithm 2 attempts to prove each bit is non-demanded by forcing its value to zero and then using the solver to

Algorithm 2 maximally precise demanded bits

```
1:  procedure computeDemandedBits(F)
2:      InputList ← getInputs(F)
3:      for each input in InputList do
4:          for i ← 0 . . . Width(input) − 1 do
5:              if askSolver(F.setBit(input, i) = F) ∧
6:                 askSolver(F.clearBit(input, i) = F) then
7:                  Result.input_i ← 0
8:              else
9:                  Result.input_i ← 1
10:     return Result
```

check if the resulting Souper expression is equivalent to the original expression. If so, we force its value to one and do another equivalence check. If both checks succeed, we have proved that the bit's value does not affect the computation, and therefore that bit is not demanded. The demanded bits abstract domain is separable following a similar argument to the one we used for known bits, and so this algorithm is also maximally precise.

3.3.3 Integer Ranges

Algorithm 3 maximally precise integer ranges

```
1:  procedure synthesizeBase(F, C_0)
2:      return constantSynthesis(
3:          if X + C_0 ≤ UINT_MAX then          ▷ Regular range
4:              F ∈ [X, X + C_0)
5:          else                                ▷ Wrapped range
6:              F ∈ [X, UINT_MAX) ∪ [0, X + C_0 − UINT_MAX)
7:      )
8:  procedure computeIntegerRange(F)
9:      L ← 1
10:     R ← UINT_MAX
11:     while L ≤ R do                          ▷ Binary Search
12:         M ← L + (R - L) / 2
13:         temp ← synthesizeBase(F, M)
14:         if temp.success then
15:             X_1 ← temp
16:             C_1 ← M
17:             R ← M - 1
18:         else
19:             L ← M + 1
20:     return [X_1, X_1 + C_1)
```

To find the most precise integer range, we are searching for a range $[X, X+C)$ that excludes more integer values than any other dataflow result. In other words, $[X, X + C)$ is a sound dataflow fact but no sound result exists for any $[X_2, X_2 + C_2)$ where $C_2 < C$. Unfortunately, the integer range abstract domain is not made of conveniently separable smaller parts, and so finding the most precise result is a more difficult problem that the ones we have faced so far.

The smallest feasible value of C can be found using binary search. Since we are not aware of a direct method for efficiently finding an X corresponding to each choice of C, we find X using synthesis. The synthesizeBase() function in Algorithm 3 attempts to find an X that, combined with a choice of C, results in a sound integer range. It uses a CEGIS-style loop to synthesize the constant, using *generalization by substitution* [12] to rule out classes of choices of X that do not work in each iteration of this loop. This function guides the binary search for the smallest C that is implemented in computeIntegerRange().

The proof of maximal precision for Algorithm 3 follows from the fact that—assuming synthesis succeeds—it always finds an integer range excluding the most possible values from the concretization set.

4 Results

4.1 Precision Comparison

We compared the precision of our algorithms against LLVM's dataflow analyses for every Souper expression encountered while compiling the C and C++ programs in version 1.0.1 of the SPEC CPU 2017 benchmark suite: 269, 113 in all. Each Souper expression has a single output (where forward analysis results end up), but may have many inputs (where the backwards dataflow information ends up). Hence, the total number of comparisons for demanded bits is over 2.1 million variables. The average Souper expression in our experiments contains 98 instructions (Souper instructions are mostly isomorphic to LLVM instructions). The largest expression is 3, 665 Souper instructions. Our implementation is based on the latest version of Souper as of May 28, 2019[2] and LLVM 8.0. SMT queries were answered using Z3 [11]; individual solver queries were timed out after 30 seconds. We also timed out any dataflow computation for a Souper expression that required more than five minutes of total execution time. Because the evaluation was time-consuming, we ran it across several machines of generally comparable (per core) power: a six-core Intel i7, a 32-core Threadripper 2, and a machine with two 28-core Xeon processors.

Table 1 summarizes the results of our precision experiment. In many cases, LLVM's analyses and ours have the same precision. This reflects the fact that LLVM's static analyses are quite good: their precision has been gradually improved and tweaked over a number of years. In a substantial minority of cases, our result is more precise than LLVM's. Each such example indicates an opportunity to improve the precision of an analysis in LLVM; we present some examples in the next section. We found no new soundness bugs in LLVM 8.0; this kind of bug would be signaled by LLVM

[2]Commit hash: f80c7990ef83d0eec529deabbbbd53a08929cb04

Table 1. Comparing the precision of LLVM's dataflow analyses and our dataflow algorithms. The "resource exhaustion" category includes cases where the solver timed out, where the solver used too much RAM, and where Souper's constant synthesis procedure fails.

Dataflow analysis	Same precision	Souper is more precise	LLVM is more precise	Resource exhaustion	Avg. CPU time per expression
Known bits	227,728 (84.6%)	17,722 (6.6%)	0	23,663 (8.8%)	9.0s
Sign bits	227,709 (84.6%)	20,920 (7.8%)	0	20,484 (7.6%)	3.2s
Non-zero	234,833 (87.3%)	13,594 (5.1%)	0	20,686 (7.7%)	2.9s
Negative	248,172 (92.2%)	1,364 (0.5%)	0	19,577 (7.3%)	2.8s
Non-negative	234,699 (87.2%)	13,749 (5.1%)	0	20,665 (7.7%)	3.3s
Power of two	247,209 (91.8%)	1,278 (0.5%)	0	20,615 (7.7%)	3.1s
Integer range	131,081 (48.7%)	22,300 (8.3%)	0	115,575 (42.9%)	16.0s
Demanded bits	741,940 (35.0%)	179,485 (8.4%)	0	1,195,368 (56.5%)	26.6s

computing a dataflow result that is "more precise" than our maximally precise result.

The right-most column of Table 1 confirms that our precise dataflow implementations are too slow to use inside a compiler. They are suitable only as an offline test oracle. Additionally, we ran into many timeout issues while computing integer ranges and demanded bits: both of these algorithms failed to compute a result around half of the time. We spent some time investigating these failures and it appears that the queries being posed are simply difficult for Z3 to handle.

4.2 Examples of LLVM Imprecisions

This section shows some examples where LLVM's dataflow analyses return imprecise results. All of these code fragments originated in SPEC CPU 2017, but in some cases we have reduced bitwidths to make the examples easier to understand. For each example, we present a fragment of LLVM-like code, followed by the maximally precise dataflow result and also LLVM's dataflow result.

4.2.1 Known Bits

Here, the value 32, represented as an 8-bit integer, is shifted left by a variable amount `%x`:

```
%0 = shl i8 32, %x

Precise %0: xxx00000
LLVM    %0: xxxxxxxx
```

Our known bits algorithm recognizes that any trailing zeroes in the original number cannot be eliminated by a left-shift operation, but LLVM returns a completely unknown result (indicated by the "x" in each bit position).

Here a four-bit value `%x` is zero-extended to eight bits and then right-shifted by a variable amount `%y`:

```
%0 = zext i4 %x to i8
%1 = lshr i8 %0, %y

Precise %1: 0000xxxx
LLVM    %1: xxxxxxxx
```

Our algorithm recognizes that high zeros cannot be destroyed by a logical right shift, whereas LLVM's analysis returns an all-unknown result.

Here, LLVM misses a known bit at the low end of a word:

```
%0 = and i8 1, %x
%1 = add i8 %x, %0

Precise %1: xxxxxxx0
LLVM    %1: xxxxxxxx
```

Its transfer function for adding known bits fails to recognize that the low bits of `%x` and `%0` are either both cleared or both set, forcing the low bit of their sum `%1` to be cleared.

When a multiplication by 10 is not allowed to overflow (the `nsw` qualifier makes signed overflow undefined) its result must be evenly divisible by 10, allowing an analysis to determine that all bits in the resulting value are zero:

```
%0 = mul nsw i8 10, %x
%1 = srem i8 %0, 10

Precise %1: 00000000
LLVM    %1: xxxxxxxx
```

LLVM's known bits computation takes advantage of integer range information, if available, but misses the fact that adding one to an integer in the range 0..4 cannot affect any bit beyond the third:

```
%x = range [0,5)
%0 = add i8 1, %x

Precise %0: 00000xxx
LLVM    %0: 0000xxxx
```

4.3 Power of Two Analysis

This section shows three LLVM fragments whose result is a power of two, but where LLVM's power-of-two dataflow analysis fails to derive that fact.

A value that is constrained to be either one or two is clearly a power of two:

```
%x = range [1,3)
```

The idiom $x \ \& \ -x$ is a commonly-used way to rapidly isolate the right-most set bit in a word. LLVM's power of two analysis recognizes this idiom, but does not connect it with range information specifying that the value cannot be zero:

```
%x = range [1,0)
%0 = sub i64 0, %x
%1 = and i64 %x, %0
```

Here a value that LLVM can easily prove to be a power of two, `%1`, is truncated; LLVM conservatively drops the power-of-two fact on the assumption that the one bit may be chopped off, failing to recognize that the constrained shift amount makes that impossible:

```
%0 = and i32 7, %x
%1 = shl i32 1, %0
%2 = trunc i32 %1 to i8
```

4.4 Demanded Bits

Recall that demanded bits is a backwards analysis; in these examples dataflow facts will be presented for the inputs, not the result, of an LLVM fragment. If the demanded bits analysis returns zero, then the value in that bit position provably does not matter.

We found many variations on this theme, where some bits of a value feeding a comparison were not demanded:

```
%0 = icmp slt i8 %x, 0

Precise demanded bits for %x: 10000000
LLVM demanded bits for    %x: 11111111
```

We also found many variations of this theme, where some bits of a value feeding an unsigned division were not demanded:

```
%0 = udiv i16 %x, 1000

Precise demanded bits for %x: 1111111111111000
LLVM demanded bits for    %x: 1111111111111111
```

4.5 Integer Range Analysis

Recall LLVM's integer range representation from Section 2.2, which is closed with respect to the lower bound and open with respect to the upper bound, and which supports wrapped ranges.

The `select` instruction is LLVM's ternary conditional expression, analogous to the `?:` construct in C and C++. The expression below returns one if `%x` is zero, and returns `%x` otherwise. Thus, the value zero is excluded from the result set:

```
%0 = icmp eq i32 0, %x
%1 = select i1 %0, i32 1, %x

Precise range for %1: [1,0)
LLVM range for    %1: full set
```

In this example, it is clear that any value in the range 1..6 will be unchanged by anding with $0xFFFFFFFF$, but LLVM pessimizes the resulting integer range slightly anyway:

```
%x = range [1,7)
%0 = and i32 4294967295, %x

Precise range for %0: [1,7)
LLVM range for    %0: [0,7)
```

The signed remainder of a 32-bit integer with eight is always in the range −7..7. LLVM's result −8..7 is pessimistic:

```
%0 = srem i32 %x, 8

Precise range for %0: [-7,8)
LLVM range for    %0: [-8,8)
```

The unsigned division operator is also analyzed imprecisely:

```
%0 = udiv i64 128, %x

Precise range for %0: [0,129)
LLVM range for    %0: full set
```

4.6 Impact of Maximal Precision on Code Generation

One might ask: What effect would increasing the precision of LLVM's dataflow analyses have on the quality of its generated code? To investigate this question, we created a version of LLVM 8.0 that has maximally precise forward bit-level dataflow analyses by invoking our algorithms. (We did not modify LLVM to call our demanded bits or integer range implementations.)

We compiled several smallish applications—gzip, bzip2, SQLite,[3] and Stockfish[4]—using LLVM 8.0 and also our modified LLVM 8.0, and compared their performance. We used the compiler flags that each application intended to use for the release (or default, if there was only one) build mode. We used "-O3" to compile gzip, bzip2, SQLite, and "-O3 -fno-exceptions -msse -msse3 -mpopcnt" to compile Stockfish. Compilation time ranged from about an hour for bzip2 to more than 70 hours for SQLite. (We would have liked to test SPEC CPU 2017 in this fashion, but compile times for its larger applications such as GCC were exceedingly long.)

To test the performance of gzip and bzip2, we compressed the 2.9 GB ISO image for SPEC CPU 2017, and decompressed the resulting compressed file. To test Stockfish (a chess engine) we ran a single-threaded benchmark computing the next move in 42 chess games that are part of its test suite, using a depth of 26 and hash table size 1024. To test SQLite (an embedded database) we ran a workload suggested in its documentation: we create a SQL database with 2,500,000

[3] https://www.sqlite.org/index.html
[4] https://stockfishchess.org/

insertions using a transaction and then run 100 selects on it.[5]

We measured performance on two machines with diverse microarchitectures: an AMD Threadripper 2990WX and an Intel Core i7-5820K. Both machines were idle except for a single core running our benchmarks. We ran each test five times and report the average execution time.

Table 2 summarizes the results of our experiment. The most interesting data point is the speedup of the compression side of bzip2. We looked at the differences between the baseline and precise versions of this program. At the level of LLVM IR, there were only a few minor differences, indicating that for this program, the middle-end optimizers did not benefit much from increased dataflow analysis precision. However, the executables were different in hundreds of locations because the SelectionDAG pass in the x86-64 LLVM backend makes heavy use of known bits information. The baseline and precise versions of bzip2 both executed roughly the same number of instructions during compression, but the instructions per cycle was considerably higher for the precise version.

4.7 Finding Soundness Errors

Since we did not find any new soundness errors in the dataflow analyses from LLVM 8.0 that we looked at, we wanted to make sure that our method can in fact detect soundness bugs. To do this, we looked for soundness bugs that had been previously fixed in LLVM, and selected three of them that had reasonably isolated patches, allowing us to port them forward to LLVM 8.0. We then used the regression tests attached to the three patches fixing these bugs to ensure that our artificially broken LLVM properly exhibits each soundness bug.

Soundness Bug 1 An LLVM patch [2] introduced many dataflow improvements including an unsound one which leads LLVM to believe that a value is provably non-zero if it is the sum of two provably non-negative values. Of course that is wrong: both non-negative values may be zero. This bug was first fixed partially and then completely in a pair of commits [3, 4].

When we ran the Souper expressions harvested from SPEC CPU 2017 through our precision checking tool, it detected this soundness bug by producing this output:

```
%0:i32 = add 0:i32, 0:i32
infer %0

non-zero from our tool: false
non-zero from llvm: true
llvm is stronger
```

For this trivial example, the buggy LLVM infers that the sum is non-zero. "llvm is stronger" means that LLVM was able to compute a stronger dataflow result than the one our algorithm computed, which is maximally precise. This is how our tool points out a soundness bug.

Soundness Bug 2 A miscompilation bug [18] in LLVM had a dataflow unsoundness bug as its root cause. The bug was fixed in patch [10]. The problem is in analysis of the signed remainder operation where the divisor is constant. This code was also triggered using one of the Souper expressions harvested by compiling SPEC:

```
%0:i32 = var
%1:i32 = srem %0, 3:i32
infer %1

known sign bits from our tool: 30
known sign bits from llvm: 31
llvm is stronger
```

Here, LLVM incorrectly believes that the signed remainder of an arbitrary 32-bit value and 3 has 31 sign bits, which is one more than it actually has.

Soundness Bug 3 This wrong code bug [21] was again in the code handling the signed remainder operation, but this time in the "known zero" analysis. It was fixed in a patch [5]. The example that triggered this bug was originally in 64 bits, but to improve readability we present an 8-bit version of it here:

```
%0:i8 = var
%1:i8 = srem 4:i8, %0
infer %1

known bits from our tool: 00000x0x
known bits from llvm: 00000x00
llvm is stronger
```

This bug was not triggered by any Souper expression found when compiling SPEC CPU 2017. However, we did trigger it using a collection of Souper expressions harvested by compiling a number of programs randomly generated by Csmith [24] and Yarpgen [1]. Given these facts, it is perhaps unsurprising that the miscompilation error in LLVM that this bug caused had been originally discovered by Csmith.

4.8 Concrete Improvements to LLVM

Some time ago (prior to the LLVM 5.0 release) we performed an early solver-based study of just the known bits analysis in LLVM, and worked with LLVM developers to get some of the worst imprecisions fixed. Some of the patches discussed below were written by us, others were written by LLVM developers.

(1) Evaluating $x \wedge (x-1)$ results in a value that always has the bottom bit cleared. The patch added to LLVM handles a slightly generalized pattern $x \wedge (x-y)$ where y must be odd. (Note: In this expression, $(\wedge)$ is used for bitwise conjunction operation, and $(\vee)$ is used for bitwise disjunction operation.)

[5] https://www.sqlite.org/speed.html

Table 2. Evaluation of the impact of maximally precise dataflow facts on generated code. The baseline compiler is LLVM 8.0 and the precise compiler is LLVM 8.0 modified to use our dataflow algorithms instead of its own.

Benchmark	AMD machine			Intel machine		
	Baseline	Precise	Speedup	Baseline	Precise	Speedup
bzip2 compress time	333.61s	260.40s	+21.94%	384.12s	353.00s	+8.10%
bzip2 decompress time	148.82s	150.94s	-1.42%	163.81s	164.02s	-0.13%
gzip compress time	75.41s	75.33s	+0.11%	82.22s	84.57s	-2.85%
gzip decompress time	14.09s	14.25s	-1.13%	14.20s	14.32s	-0.84%
Stockfish total time	257.28s	254.68s	+1.01%	273.99s	272.54s	+0.53%
SQLite	37.78s	36.78s	+2.65%	40.01s	39.88s	+0.32%

(2) The LLVM byte-swap intrinsic function, used to change the endianness of a value, was not previously handled by the known bits analysis. This is fixed now.
(3) The additive inverse of a zero-extended value, $0 - \text{zext}(x)$, is always negative.
(4) The result of the Hamming weight intrinsic in LLVM (`@llvm.ctpop`) had room for improvement.
(5) Tests for equality and inequality can sometimes be resolved at compile time if, for example, $x = y$ is being evaluated and at some bit position, x is known to have a zero and y is known to have a one.

Manually identifying imprecisions in dataflow analyses is difficult. Given some code to compile, we can identify imprecisions automatically. In this section we showed that five such issues in LLVM have already been fixed based on the results of our work.

5 Related Work

Our work fits into the broader context of compiler testing techniques, about which much has been written [17]. Compilers are large, diverse programs and some of their parts can benefit from specific testing. In this section we focus on methods for testing static analyses.

Bugariu et al. [6] test numerical abstract domains by creating a collection of representative abstract values, applying a sequence of abstract transfer functions, and then testing the results using a collection of properties that check for unsoundness, imprecision, or failure to converge. They found bugs in libraries such as Apron. This work has the same goals as our work, but uses a very different test oracle: theirs is based on high-level properties and ours is based on computing sound and precise results using algorithms that call out to an SMT solver.

Randomized differential testing has been used to find soundness and precision issues in static analyzers. Cuoq et al. [8] instrumented an analyzer with assertions to compare the inferred values with the values obtained from concrete execution. Klinger et al. [16] used differential testing to evaluate both the soundness and precision of program analyzers. Their work tested six different program analyzers and found defects in four of them. Klinger et al.'s goals are the same as ours, but we avoided differential testing for two reasons. First, there do not exist any other implementations of most of the static analyses for LLVM IR that we target. Second, our solver-based techniques are maximally precise, avoiding the limitation that differential testing cannot, in general, find imprecisions in the most precise tool being tested.

Midtgaard and Møller [20] used ideas from QuickCheck [7] to test properties of static analyses using a domain-specific language to specify properties to be tested.

Madsen et al. [19] verify the correctness and safety of abstract domains implemented in static analysis tools. Their technique uses SMT solvers and is specifically designed for Flix, a functional and logic programming language designed for implementing static analyses. This technique allows a user to define a lattice and transfer functions in a functional language that is intended to be verified. For this to work, they extended Flix to support annotations and laws to specify properties of abstract domains to be tested. They verify abstract domains using an SMT solver and also support testing based on QuickCheck [7, 20].

6 Conclusion

Dataflow analyses are a key enabling technology for optimizing compilers. In this paper, we presented a collection of novel solver-based algorithms for computing maximally precise data flow analysis results. These results made it possible to test dataflow implementations in LLVM for soundness and precision. We found numerous imprecisions, some of which have been fixed as a result of our work.

A Artifact Appendix

A.1 Abstract

This artifact provides the source code that implements our work on testing precision and soundness of LLVM's dataflow analyses. Our work builds on the open source superoptimizer, Souper [22] and evaluates LLVM 8.0. The benchmarks used in our experiments are open-source, except the SPEC CPU 2017 benchmark that has restricted access and cannot be distributed publicly. We provide a Dockerfile in our artifact

to automatically build the Docker image, which takes care of installing all required dependencies, benchmark suite, and Souper for the experimental evaluation.

A.2 Artifact Checklist

- **Algorithm:** SMT solver-based algorithms to compute precise dataflow facts
- **Program:** Souper: an open-source superoptimizer for LLVM IR [22], Z3 solver, re2c, Redis, benchmarks: Gzip, Bzip2, Stockfish, SQLite, SPEC CPU 2017 ISO image v1.0.1.
- **Compilation:** Modern compiler toolchain: GCC-7.4 and above or Clang-7.0 and above.
- **Run-time environment:** Experiments are evaluated on Linux Ubuntu 18.04 for x86-64.
- **Hardware:** Commodity x86-64 machines.
- **Experiments:** Build the Docker image, run the image, and follow the instructions to run the scripts to validate the results.
- **Output:** Scripts are provided to process the raw data (in textual format) generated by experiments. We have also provided instructions for expected results and analysis of the final output.
- **How much disk space required (approximately)?:** 20 GB disk space.
- **How much time is needed to prepare workflow?:** Building the docker image takes about an hour.
- **How much time is needed to complete experiments?:** It takes approximately an hour to complete all experiments except precision testing of the SPEC CPU 2017 benchmark, which may take up to 100 hours to test eight dataflow facts.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0, University of Illinois/NCSA Open Source License.

A.3 Description

A.3.1 Distribution

The artifact is publicly available and can be downloaded from URL: https://doi.org/10.1145/3373122. We provide scripts that take care of building the prerequisites, our tool, and running the experiments.

A.3.2 Hardware Dependencies

We recommend testing on a modern Linux machine, with a modern compiler toolchain like GCC/Clang. We used Ubuntu 18.04 with GCC 7.4 for our work.

A.3.3 Software Dependencies

Our work is implemented in C++, and the scripts are written in Perl, Python, and Bash. Our tool uses LLVM+Clang 8.0 compiler, Z3 solver, Redis database to cache the results, open-source benchmarks (bzip2, gzip, SQLite, Stockfish) for evaluation. The script *Dockerfile* in the artifact's archive takes care of building all required software dependencies in the Docker image.

A.4 Installation

Install the Docker engine by following the instructions from the URL: https://docs.docker.com/install/linux/docker-ce/ubuntu/.

Download our artifact from the archive and untar it.

```
$ tar -xf souper-cgo20-artifact.tar.gz
$ cd souper-cgo20-artifact
```

Now, you can build and run the Docker image.

```
$ ./build_docker.sh
$ sudo docker run -it jubitaneja/artifact-cgo \
    /bin/bash
$ export PS1="(docker) $PS1"
```

The entire setup of our tool in docker is at this path:

```
(docker) $ cd /usr/src/artifact-cgo
```

A.5 Experiment Workflow

The artifact provides the scripts to reproduce the results presented in Section 4.

Precision Testing of SPEC CPU Benchmark: We do not share the SPEC ISO image because of its restricted license agreement, and instead, provide the Redis database file *(dump.rdb.7z)* of input Souper expressions collected from building SPEC CPU benchmark using Souper.

The *SPEC.md* file in the archive provides detailed instructions on how to setup a Redis server for testing each dataflow fact and reproduce the results, as shown in Table 1 and details in Section 4.1.

Precision Testing of Individual Dataflow Facts: The script *test_precision.sh* compares the dataflow facts computed by our precise algorithms with the LLVM compiler. It tests each example discussed in Section 4.2 to Section 4.5. You can run the script in the docker setup.

```
(docker) $ cd /usr/src/artifact-cgo/precision/test
(docker) $ ./test_precision.sh
```

Performance Evaluation: The script *test_performance.sh* measures the impact of the precision of dataflow analysis for applications: Bzip2, Gzip, Stockfish, and SQLite. The script reproduces the results, as presented in Table 2 and discussed in Section 4.6. You can run the script:

```
(docker) $ cd /usr/src/artifact-cgo/performance/test
(docker) $ ./test_performance.sh
```

Soundness Testing: The script *test_sound.sh* tests three soundness bugs discussed in Section 4.7. You can run the script in the docker setup.

```
(docker) $ cd /usr/src/artifact-cgo/soundness/test
(docker) $ ./test_sound.sh
```

A.6 Evaluation and Expected Results

The scripts mentioned in the previous section generate the output on the standard console or redirect it to the files. You can compare the results to the data presented in Section 4.

We evaluate the performance of compression applications by compressing and decompressing the SPEC CPU benchmark ISO image, a 2.9 GB file, in Section 4.6. However, we cannot share the proprietary source and thus modified the setting of an experiment for the artifact. In the artifact, we provide a randomly generated 1 GB file using *dd* utility for the evaluation of Bzip2 and Gzip.

The detailed analysis of each result and the expected output is discussed in *README.md* file in the archive in "Section 3: Analysis of the Results".

A.7 Experiment Customization

You can easily customize test inputs written in the Souper IR, and try different options to compute the dataflow facts from maximally precise algorithms and compare it with the facts computed by the LLVM compiler. The list of options to compute dataflow information from the maximally precise algorithms that we have implemented in Souper is:

- Known bits: -infer-known-bits
- Sign bits: -infer-sign-bits
- Negative DFA: -infer-neg
- Non-negative DFA: -infer-non-neg
- Non-zero DFA: -infer-non-zero
- Power of two DFA: -infer-power-two
- Range DFA: -infer-range -souper-range-max-precise -souper-range-max-tries=300
- Demanded bits: -infer-demanded-bits

A sample command line to compute known bits for an input file, *input.opt* written in Souper IR is:

```
$ souper-check -infer-known-bits \
    -z3-path=/path/to/z3 input.opt
```

On the other side, while computing dataflow facts from the LLVM compiler using Souper, which calls LLVM's dataflow functions, the different options are:

- Known bits: -print-known-at-return
- Sign bits: -print-sign-bits-at-return
- Negative DFA: -print-neg-at-return
- Non-negative DFA: -print-nonneg-at-return
- Non-zero DFA: -print-non-zero-at-return
- Power of two DFA: -print-power-two-at-return
- Range DFA: -print-range-at-return
- Demanded bits: -print-demanded-bits-from-harvester

A sample command line to compute known bits from the LLVM compiler for an input file, *input.opt* written in Souper IR is:

```
$ souper2llvm input.opt | llvm-as | souper \
    -print-known-at-return
```

'souper2llvm' is a utility to translate an input in Souper IR to the LLVM IR. 'llvm-as' is the LLVM's assembler to generate the LLVM bitcode.

The details of customizing the experiments are discussed in "Section 4: Experiment Customization" in *README.md* file in the archive.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1218022 and the Office of Naval Research under Grant No. N00014-17-1-2996.

References

[1] Dmitry Babokin, Anton Mitrokhin, and Vsevolod Livinskiy. 2017. Yarpgen: Yet Another Random Program Generator. https://github.com/intel/yarpgen.

[2] Baldrick. 2011. Revision 124183. http://llvm.org/viewvc/llvm-project?view=revision&revision=124183. [Online; accessed 08-27-2019].

[3] Baldrick. 2011. Revision 124184. http://llvm.org/viewvc/llvm-project?view=revision&revision=124184. [Online; accessed 08-27-2019].

[4] Baldrick. 2011. Revision 124188. http://llvm.org/viewvc/llvm-project?view=revision&revision=124188. [Online; accessed 08-27-2019].

[5] Baldrick. 2012. Revision 155818. http://llvm.org/viewvc/llvm-project?view=revision&revision=155818. [Online; accessed 08-27-2019].

[6] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 768–778.

[7] Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices* 46, 4 (2011), 53–64.

[8] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods Symposium*. Springer, 120–125.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (Oct. 1991), 451–490.

[10] Sanjoy Das. 2015. Revision 233225. http://llvm.org/viewvc/llvm-project?view=revision&revision=233225. [Online; accessed 08-27-2019].

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

[12] Bruno Dutertre. 2015. Solving Exists/ForAll Problems With Yices. In *Workshop on satisfiability modulo theories*.

[13] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2013. Abstract Interpretation over Non-Lattice Abstract Domains. In *International Static Analysis Symposium*. Springer, 6–24.

[14] Yamauchi Hiroshi. 2017. [LVI] Constant-propagate a zero extension of the switch condition value through case edges. https://reviews.llvm.org/rL309415. [Online; accessed 11-12-2017].

[15] Uday P Khedker and Dhananjay M Dhamdhere. 1994. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1472–1511.

[16] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2018. Differentially Testing Soundness and Precision of Program Analyzers. *arXiv preprint arXiv:1812.05033* (2018).

[17] Alexander S Kossatchev and MA Posypkin. 2005. Survey of Compiler Testing Methods. *Programming and Computer Software* 31, 1 (2005), 10–19.

[18] Nick Lewycky. 2015. Miscompile of % in loop. https://bugs.llvm.org/show_bug.cgi?id=23011. [Online; accessed 08-27-2019].

[19] Magnus Madsen and Ondřej Lhoták. 2018. Safe and Sound Program Analysis with FLIX. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 38–48.

[20] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Software Testing, Verification and Reliability* 27, 6 (2017), e1640.

[21] John Regehr. 2012. Wrong code bug. https://bugs.llvm.org/show_bug.cgi?id=12541. [Online; accessed 08-27-2019].

[22] John Regehr, Raimondas Sasnauskas, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and Yang Chen. 2017. Souper: A Synthesizing Superoptimizer. https://arxiv.org/abs/1711.04422. https://github.com/google/souper.

[23] Wikipedia-Semilattice. 2019. Semilattice — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Semilattice [Online; accessed 01-09-2019].

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 283–294.