Automatically Testing Implementations of Numerical Abstract Domains

Alexandra Bugariu
Department of Computer Science, ETH Zurich
Switzerland
alexandra.bugariu@inf.ethz.ch

Valentin Wüstholz
Department of Computer Science, ETH Zurich
Switzerland
wuestholz@gmail.com

Maria Christakis
MPI-SWS
Germany
maria@mpi-sws.org

Peter Müller
Department of Computer Science, ETH Zurich
Switzerland
peter.mueller@inf.ethz.ch

ABSTRACT

Static program analyses are routinely applied as the basis of code optimizations and to detect safety and security issues in software systems. For their results to be reliable, static analyses should be sound (i.e., should not produce false negatives) and precise (i.e., should report a low number of false positives). Even though it is possible to prove properties of the *design* of a static analysis, ensuring soundness and precision for its *implementation* is challenging. Complex algorithms and sophisticated optimizations make static analyzers difficult to implement and test.

In this paper, we present an automatic technique to test, among other properties, the soundness and precision of abstract domains, the core of all static analyzers based on abstract interpretation. In order to cover a wide range of test data and input states, we construct inputs by applying sequences of abstract-domain operations to representative domain elements, and vary the operations through gray-box fuzzing. We use mathematical properties of abstract domains as test oracles. Our experimental evaluation demonstrates the effectiveness of our approach. We detected several previously unknown soundness and precision errors in widely-used abstract domains. Our experiments also show that our approach is more effective than dynamic symbolic execution and than fuzzing the test inputs directly.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

soundness testing, precision testing, abstract interpretation

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240464

ACM Reference Format:
Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3240464

1 INTRODUCTION

Static program analyses compute semantic properties of programs, which are the basis for program optimizations and for detecting program errors and security vulnerabilities. Since most properties of programs are undecidable, static analyses approximate the set of possible program behaviors. For its results to be reliable, a static analysis should be sound and precise. A *sound* analysis considers each possible program behavior, that is, computes an over-approximation of all possible behaviors. Consequently, sound analyses do not produce false negatives. A *precise* analysis computes a tight approximation to minimize the number of false positives.

Many static analyses are based on the abstract-interpretation framework [20]. In this framework, abstractions of the program state are represented by elements of *abstract domains*; for instance, numerical abstract domains may track intervals of possible values for numerical variables or constraints between them. The semantics of program operations is represented by *abstract transformers*, which specify the effect of an operation on the abstract state.

Even though it is possible to prove properties of the *design* of a static analysis, ensuring soundness and precision for its *implementation* is challenging. In fact, implementations of abstract domains are often complex and highly optimized in order to maximize performance and scalability [46]. Errors in these implementations likely affect the usefulness of all static analyzers that build on them.

Imagine a static analysis that abstracts numerical variables to intervals of possible values. For instance, the abstract value $[0, 5]$ for an integer variable x expresses that, in each program execution, the concrete (actual) value of x satisfies $0 \leq x \leq 5$. The example in Fig. 1 illustrates a potential soundness error due to arithmetic overflow. Without prior knowledge about parameter `p`, its abstract value is $[INT\_MIN, INT\_MAX]$, and consequently, the abstract value of `a` after the assignment is $[INT\_MIN + 1, INT\_MAX + 1]$. If this abstract domain is implemented using bounded integers, a naive implementation of the addition operator will lead to an overflow and

```
int foo(int p) {
  int a := p + 1;
  ...
}
```

Figure 1: Potential soundness error due to overflow.

produce $[INT\_MIN + 1, INT\_MIN]$. This empty interval indicates that the code after the assignment is unreachable, which is unsound. In particular, this unsound result might mask program errors and security vulnerabilities in subsequent code.

In this paper, we assess the effectiveness of existing automatic testing techniques in finding, among other errors, soundness and precision issues in widely-used implementations of abstract domains. We generate test inputs using a novel combination of existing ideas: starting from a pool of pre-selected values, we apply abstract-domain operations to create representative domain elements, and vary the operations by employing an off-the-shelf gray-box fuzzer, such as AFL [7] or LibFuzzer [6], to maximize coverage. As in earlier work by Midtgaard and Møller [40], we use mathematical properties of abstract domains as test oracles. However, we target real-world implementations of complex abstract domains (e.g., APRON's Octagons domain [41] and ELINA's Polyhedra domain [46]), and we extend the set of tested properties by including more precision properties and by approximating termination properties.

Our evaluation on several abstract domains of the APRON [31] and ELINA [46] libraries shows that our combination effectively detects soundness and precision problems in complex, mature implementations. In particular, we show that it is more effective than purely relying on existing test case generation techniques, such as gray-box fuzzing and dynamic symbolic execution (DSE, also known as concolic testing) [14, 28].

Our main contributions are the following:

(1) We present a novel combination of automatic test case generation techniques to detect, among other errors, soundness and precision issues in implementations of abstract domains.
(2) We demonstrate that our technique effectively finds both seeded and real errors in widely-used implementations of numerical abstract domains.
(3) We show that our technique tests abstract domains more effectively than standard DSE or gray-box fuzzing.

Our experience is useful for developers of abstract domains to ensure soundness and precision of their implementations. It is also useful for developers of static analyzers to assess the quality of available abstract-domain libraries. Even if the design of an abstract domain intentionally sacrifices soundness in favor of other qualities [16, 37], it is still important to check the implementation for *unintentional* unsoundness caused by implementation errors.

Outline. The next section summarizes some background on abstract interpreters. Sect. 3 gives an overview of our approach, and Sect. 4 explains the technical details. In Sect. 5, we present our experimental evaluation on real-world implementations of abstract domains. We discuss related work in Sect. 6 and conclude in Sect. 7.

2 BACKGROUND

Abstract interpretation [20] is a theoretical framework for expressing static analyses, used by many industrial analyzers, such as Astrée [12] and Clousot [26]. It relies on *abstract domains* to represent abstractions of concrete program states, and on *abstract transformers* to model the behavior of program instructions, such as assignments and conditionals.

Abstract domains are often reused across many different program analyses. For example, most static analyzers employ numerical domains, which are, therefore, the focus of this paper. Widely-used numerical domains include Intervals [19], which capture the range of values for each variable, Octagons [41], which can also express simple relations between two variables, Polyhedra [23], which can capture linear inequalities between arbitrarily many variables, and Zonotopes [29], which express affine relations.

Most abstract domains are represented by complete lattices of the form $(\bar{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$. $\bar{A}$ denotes the set of abstract elements $\bar{x}$, which are partially ordered by *inclusion* $\sqsubseteq$. Each abstract element represents a set of *constraints*, i.e., mathematical relations between variables and constants. The *bottom* element $\bot$ is the least element of the lattice; it represents the empty set of concrete states and corresponds to an unsatisfiable set of constraints. The *top* element $\top$ is the greatest element and represents the universal set of concrete states; that is, all variables are unconstrained.

$\sqcup$ and $\sqcap$ are the *join* and *meet* operators, which are used to obtain the union, respectively the intersection, of two abstract elements. Additionally, an abstract domain whose lattice has an infinite height requires a *widening* operator ($\triangledown$) to ensure that the analysis eventually reaches a fixed point. Some domains (such as Intervals and Octagons) also support a *narrowing* operator ($\triangle$), which can improve the precision of the analysis [22].

Abstract transformers are typically specific to a given analysis and programming language, but some transformers are universal building blocks for many analyses. These include *assign* to represent an assignment, *cond* to assume a condition to hold, and *project* to eliminate any previous information about one of the variables (e.g., when a new value of a variable is read from a file). Here, we focus on these transformers because they are the most complex to implement [47] (and thus the most error prone). A generalization of our approach to other transformers is mostly straightforward.

3 OVERVIEW

Since implementations of abstract domains are often used as libraries by many different program analyses, we apply a unit testing approach. Compared to system testing, unit testing allows us to specify generic test oracles, which are independent of a specific abstract domain or static analysis. Moreover, it facilitates the generation of test data because abstract-domain elements can be generated much more easily than input programs for a whole analyzer [40]. In this section, we give an overview of the three main ingredients of our automatic unit test approach: test oracles, test input generation, and test drivers. Details will be presented in Sect. 4.

3.1 Test Oracles

The abstract interpretation framework prescribes a number of properties of the domain operators and abstract transformers (collectively referred to as *domain operations* in the rest of the paper), which are required for soundness. For instance, if the abstract element $\bar{x}$ (capturing the pre-state of an assignment) is reachable

(that is, different from bottom), the post-state of the assignment $v := e$ should also be non-bottom for all variables v and well-formed expressions e:

$$\bar{x} \neq \bot \Rightarrow assign(\bar{x}, v, e) \neq \bot \quad \text{(a)}$$

We use these general soundness conditions as test oracles. We also identify a number of general precision conditions, whose violation indicates that the result of a domain operation is over-approximated more than necessary. For this purpose, we compare the result of a domain operation to the *best transformer*, that is, the most precise result *representable* in a given abstract domain. For instance, the result of intersecting top with an element should be equal to the element itself:

$$\top \sqcap \bar{x} = \bar{x} \quad \text{(b)}$$

Moreover, we check that widening and narrowing converge within a given number of iterations, which ensures the termination of any fixed-point computation in which they are used.

Note that all the properties considered in this paper are defined under the assumption that the analyzed programs do not raise exceptions, otherwise the behavior of the static analyzer depends on the semantics of the programming language. For example in C, division by zero causes undefined behavior. When classifying the properties into soundness and precision (as described in Sect. 4), we assume that the abstract domain does not model error states.

3.2 Input Data

Abstract domains often use sophisticated data structures to optimize performance. For instance, elements of the Polyhedra domain are typically represented using both matrices and vectors of floating-point numbers. In our experiments, we observed that the standard test case generation techniques do not work well for complex abstract domains. In particular, fuzzing failed to detect subtle interactions between domain operations, and DSE did not effectively explore real-world implementations that make heavy use of floating-point arithmetic and libraries. We, therefore, use a combination of test case generation techniques to create a pool of domain elements, which serve as test inputs to domain operations. The pool is populated in two steps.

Step 1 creates an initial pool by combining boundary and random testing. Each element of a numerical domain can be constructed from numerical constraints. For instance, an element of the Intervals domain, which maps program variables v_i to their possible ranges, is created from the constraint $k_l \leq v_i \leq k_u$ (the constants k_l and k_u are the lower and upper bounds). We generate the numerical constants randomly or choose them from a pre-defined set of boundary values that are more likely to expose bugs, such as off-by-one errors and arithmetic overflows. If the Intervals domain is implemented using machine integers, these boundary values are $\{INT\_MIN, 0, INT\_MAX\}$. The initial pool also contains the extreme elements $\top$ and $\bot$.

Selecting inputs from this pool (together with a suitable expression e) as arguments to the *assign* transformer will likely detect the possible unsoundness illustrated in Fig. 1. The initial pool is likely to contain an element mapping the variable `p` to $[0, INT\_MAX]$ since 0 and INT_MAX are pre-defined boundary values; moreover, we are likely to obtain an expression e of the from `p`$+k$ for a positive constant k (see Sect. 4 for details). Evaluating the *assign* transformer on these inputs violates soundness property (a) defined above.

The initial pool allows us to test *individual* domain operations. However, this approach is insufficient in two situations. First, some implementations rely on complex consistency conditions on their data structures. For Polyhedra, for instance, the two internal representations must be kept consistent. If a faulty operation violates this invariant, the effect can be often observed only when applying a subsequent operation; it is thus necessary to perform at least two consecutive operations to detect the bug.

Second, there are certain soundness or precision properties of *individual* operations that cannot be checked by generic, domain-independent oracles. For instance, the *assign* transformer for Octagons should, in some cases, apply a so-called closure operation; failing to do so or using an imprecise closure may lead to loss of precision in subsequent operations, such as inclusion or equality tests [41]. A test oracle that directly detects a missing closure would be specific to Octagons and, thus, not reusable. A more generic way to detect this problem is by intersecting the result of the *assign* transformer with top. The expected output of the intersection is the same as the result of the *assign* transformer itself. However, if the transformer does not apply the closure when expected, the domain-independent property (b) may fail due to imprecision.

The above situations can both be tested by executing at least two consecutive domain operations before checking the oracle. Therefore, step 2 of our input generation expands the pool of domain elements by iteratively applying a domain operation to existing pool elements (and possibly other arguments) and adding the result to the pool. By repeating this process, we increase the likelihood of constructing elements that need to be built incrementally with several domain operations. As a result, we are also more likely to detect bugs that manifest themselves only in consecutive operations.

3.3 Test Drivers and Exploration

For each property under test, we manually write a test driver that is parametric in: (1) the operations and their arguments used in step 2 to populate the pool of domain elements, and (2) the arguments of the property under test.

Fig. 2 shows, in pseudo code, our test driver for checking soundness property (a) for the *assign* transformer. The driver takes as arguments a sequence of operations (`ops`), which are used in step 2 of the pool creation, a sequence of indices into the pool (`elems`), a sequence of expressions (`exprs`), and a sequence of program variables (`vars`). The latter three provide arguments to the domain operations. The driver also takes as arguments an index into the pool (`oelem`), an expression (`oexpr`), and a variable (`ovar`), which are inputs to the property under test (or oracle) (lines 17–21).

The test driver initializes the pool of abstract elements by creating valid inputs for the currently tested domain, such as Intervals or Octagons (step 1, line 4). Then, it extends the pool by applying domain operations (step 2, lines 7–14). Each iteration obtains the next operation and its arguments from the parameters of the test driver, applies it, and adds the result to the pool. Finally, the test driver asserts property (a) after computing the result of the *assign* operation. For simplicity, we omit optimizations (for instance, our implementation initializes the pool only once, before all the drivers

```
1  test_assign(ops, elems, exprs, vars,
2                  oelem, oexpr, ovar) {
3    // populate pool, step 1
4    pool := init();
5
6    // populate pool, step 2
7    for(i := 0; i < nbops; i++) {
8      op := ops[i];
9      x := pool[elems[i]];
10     y := pool[elems[nbops + i];
11     e := exprs[i];
12     v := vars[i];
13     add_to_pool(pool, apply(op, x, y, e, v));
14   }
15
16   // execute assign transformer
17   x := pool[oelem]
18   res := apply(assign_op, x, ovar, oexpr);
19
20   // check property (a)
21   assert x ≠ ⊥ ⇒ res ≠ ⊥;
22 }
```

Figure 2: Pseudo code of a test driver for checking soundness property (a) of the assignment transformer.

	Partial order	
1	$\bot \sqsubseteq \bar{x}$	[P]
2	$\bar{x} \sqsubseteq \top$	[P]
3	$\bar{x} \sqsubseteq \bar{x}$	[P]
4	$\bar{x} \sqsubseteq \bar{y} \wedge \bar{y} \sqsubseteq \bar{z} \Rightarrow \bar{x} \sqsubseteq \bar{z}$	[P]
5	$\bar{x} \sqsubseteq \bar{y} \wedge \bar{y} \sqsubseteq \bar{x} \Rightarrow \bar{x} = \bar{y}$	[P]
	Join	
6	$\bot \sqcup \bar{x} = \bar{x}$	[P]
7	$\top \sqcup \bar{x} = \top$	[S]
8	$\bar{x} \sqsubseteq \bar{x} \sqcup \bar{y}$	[S]
9	$\bar{y} \sqsubseteq \bar{x} \sqcup \bar{y}$	[S]
10	$\bar{x} \sqcup \bar{y} = \bar{y} \sqcup \bar{x}$	[P]
11	$(\bar{x} \sqcup \bar{y}) \sqcup \bar{z} = \bar{x} \sqcup (\bar{y} \sqcup \bar{z})$	[P]
12	$\bar{x} \sqcup \bar{x} = \bar{x}$	[P]
13	$\bar{x} \sqsubseteq \bar{y} \Rightarrow \bar{x} \sqcup \bar{y} = \bar{y}$	[P]
14	$\bar{x} \sqcup \bar{y} = \bar{y} \Rightarrow \bar{x} \sqsubseteq \bar{y}$	[P]
15	$\bar{x} \sqcup (\bar{x} \sqcap \bar{y}) = \bar{x}$	[P]
	Meet	
16	$\bot \sqcap \bar{x} = \bot$	[P]
17	$\top \sqcap \bar{x} = \bar{x}$	[P]
18	$\bar{x} \sqcap \bar{y} \sqsubseteq \bar{x}$	[P]
19	$\bar{x} \sqcap \bar{y} \sqsubseteq \bar{y}$	[P]
20	$\bar{x} \sqcap \bar{y} = \bar{y} \sqcap \bar{x}$	[P]
21	$(\bar{x} \sqcap \bar{y}) \sqcap \bar{z} = \bar{x} \sqcap (\bar{y} \sqcap \bar{z})$	[P]
22	$\bar{x} \sqcap \bar{x} = \bar{x}$	[P]
23	$\bar{x} \sqsubseteq \bar{y} \Rightarrow \bar{x} \sqcap \bar{y} = \bar{x}$	[P]
24	$\bar{x} \sqcap \bar{y} = \bar{x} \Rightarrow \bar{x} \sqsubseteq \bar{y}$	[P]
25	$\bar{x} \sqcap (\bar{x} \sqcup \bar{y}) = \bar{x}$	[P]
26	$disj(\bar{x}, \bar{y}) \Rightarrow \bar{x} \sqcap \bar{y} = \bot$	[P]
	Join/Meet bounds	
27	$\forall \bar{b} : (\bar{x} \sqsubseteq \bar{b}) \wedge (\bar{y} \sqsubseteq \bar{b}) \Rightarrow (\bar{x} \sqcup \bar{y} \sqsubseteq \bar{b})$	[P]
28	$\forall \bar{b} : (\bar{b} \sqsubseteq \bar{x}) \wedge (\bar{b} \sqsubseteq \bar{y}) \Rightarrow (\bar{b} \sqsubseteq \bar{x} \sqcap \bar{y})$	[P]
	Widening	
29	$\bar{x} \sqsubseteq \bar{x} \triangledown \bar{y}$	[S]
30	$\bar{y} \sqsubseteq \bar{x} \triangledown \bar{y}$	[S]
31	$\bar{x} \triangledown \bot = \bar{x}$	[P]
32	$\bot \triangledown \bar{x} = \bar{x}$	[P]
33	widening converges	[C]
	Assignment $[a = assign]$	
34	$\bar{x} \sqsubseteq \bar{y} \Rightarrow a(\bar{x}, v, e) \sqsubseteq a(\bar{y}, v, e)$	[P]
35	$\bar{x} \neq \bot \Rightarrow a(\bar{x}, v, e) \neq \bot$	[S]
36	$\bar{x} = \bot \Rightarrow a(\bar{x}, v, e) = \bot$	[P]
37	$rep(e, \bar{x}) \Rightarrow a(\bar{x}, v, e) \neq \top$	[P]
	Projection $[p = project]$	
38	$a(\bar{x}, v, e) \sqsubseteq p(\bar{x}, v)$	[P]
	Conditional $[c = cond]$	
39	$\bar{x} \sqsubseteq \bar{y} \Rightarrow c(\bar{x}, e) \sqsubseteq c(\bar{y}, e)$	[P]
40	$\bar{x} = \bot \Rightarrow c(\bar{x}, e) = \bot$	[P]
41	$c(\bar{x}, e) \sqsubseteq \bar{x}$	[P]
	Narrowing	
42	$\bar{x} \sqcap \bar{y} \sqsubseteq \bar{x} \vartriangle \bar{y}$	[P]
43	$\bar{x} \vartriangle \bar{y} \sqsubseteq \bar{x}$	[P]
44	$\bar{x} \vartriangle \bot = \bot$	[P]
45	$\bot \vartriangle \bar{x} = \bot$	[P]
46	narrowing converges	[C]

Figure 3: Algebraic properties of abstract domains. We classify them into soundness [S], precision [P], or convergence [C]. Note that all free variables are implicitly universally quantified and all variables refer to abstract-domain elements except for variables v and e, which refer to a program variable and an expression, respectively. Predicate $disj(\bar{x}, \bar{y})$ denotes that the intersection of the set of constraints from $\bar{x}$ and $\bar{y}$ is trivially empty, and predicate $rep(e, \bar{x})$ that e can be precisely represented in the abstract domain of $\bar{x}$.

are run) and several technicalities, such as ensuring that indices are within bounds.

We use the state-of-the-art gray-box fuzzer LibFuzzer [6] to provide the arguments to the test driver, i.e., to choose the operations used in step 2 together with their arguments, and the parameters for the property under test. Gray-box fuzzers use a lightweight code instrumentation to generate inputs that are likely to execute previously uncovered code. In our implementation, all parameters of the test driver are encoded into one array, which is created by the fuzzer.

4 TECHNICAL SOLUTION

In this section, we provide the technical details of our approach.

4.1 Test Oracles

Our test oracles are based on domain-independent mathematical properties of abstract operations. In the following, we give an overview of these properties and explain how they are checked by the test drivers.

Properties. Based on the abstract-interpretation literature [20, 21] and earlier work on testing static analyzers [40], we identified 46 properties that need to be satisfied by domain operations to ensure soundness, precision, and termination. Fig. 3 provides an overview.

The soundness properties are required to ensure that an abstract-domain element over-approximates the concrete states it represents. We have discussed an example in the previous section (property 35).

To deal with the undecidability of most semantic program properties, static analyses over-approximate the set of concrete program behaviors and then infer or check properties on this abstraction. Since they are intended to compute an approximation, one cannot expect the operations of an abstract domain to be precise w.r.t. the concrete program execution. Therefore, our precision properties check that the domain operations do not lead to *unnecessary* information loss, that is, they compare the result of an operation to the most precise representable result as obtained by applying the best transformer (see Sect. 3).

For instance, computing the join of two intervals $[1, 1]$ and $[3, 3]$ yields $[1, 3]$, which loses the information that the variable is different from 2. Despite this inevitable information loss, a join operator should satisfy a number of precision properties (6, 10–13, 15); for instance, property 13 prevents the join of $[1, 1]$ and $[1, 3]$ from returning $[0, 4]$ or $\top$, which is sound, but unnecessarily imprecise.

The soundness and convergence properties need to hold for all abstract domains; some precision properties may not always hold when the best transformers do not exist or cannot be computed [43]. For instance, domains based on incomplete lattices (such as Zonotopes) do not have a least upper bound for every pair of abstract elements. This can force $\sqcup$ to return a larger upper bound and, thus, violate property 27. For such domains, we require developers to select the subset of precision properties that should be checked.

The convergence properties (33 and 46) require widening and narrowing to eventually reach a fixed point, which is necessary to ensure that the static analysis of loops and recursion terminates. Since convergence is a termination property, which cannot be tested, we instead check the stronger property that a fixed point is reached within a given number of iterations, as we explain below.

Executable oracles. We manually construct a test driver for each of the properties in Fig. 3. This driver selects suitable inputs for

```
1  i := 0;
2  x := pool[oelems[i]];
3  while (true) {
4    i := i + 1;
5    y := pool[oelems[i]];
6    res := apply(widening_op, x, y);
7    if (res == x)
8      break; // a fixed point is reached
9    x := res;
10   assert i < k;
11 }
```

Figure 4: Fragment of a test driver for checking whether the Octagons widening reaches a fixed point within k steps.

each of the free variables of the property to be tested, evaluates the property, and checks that it holds. For instance, for property 3, it selects a domain element d for variable $\bar{x}$ and checks that the $\sqsubseteq$ operator yields true when applied to d and d.

Translating properties into executable oracles is straightforward for most soundness and precision properties, but slightly more involved for convergence properties. Fig. 4 shows a fragment of our test driver for checking whether the Octagons widening converges after k iterations [41]. The driver computes an increasing chain of x elements (as checked by property 29), each obtained by widening the previous element with an arbitrary element y. Widening converges if the x-chain becomes stable, here, within k steps. We observed that $k = 100$ is a sufficient upper bound for all our tested analyzers, because widening converges much faster in practice.

Note that, for most abstract domains (such as Intervals, Octagons, etc.), widening can be applied to arbitrary elements. There are, however, some exceptions; for instance, the Polyhedra widening requires monotonicity of its operands (that is, $x \sqsubseteq y$). In such cases, we use a slightly different test driver that applies widening on x and the join of x and y; that is, by changing the last parameter of `apply` on line 6 of Fig. 4 to $x \sqcup y$, because $x \sqsubseteq x \sqcup y$ (see property 8). As a consequence of this monotonicity precondition, property 31 needs to hold for the Polyhedra domain only for $\bar{x} = \bot$.

4.2 Input Data

Testing the properties from Fig. 3 requires three kinds of input data: (1) program variables, (2) expressions over them (for the *assign* and *cond* transformers), and (3) abstract-domain elements that contain constraints over these variables. We construct this data as follows.

Program variables. All our test cases operate on a set of pre-defined integer variables. The number of variables must be sufficiently small to keep the memory consumption and execution time of the test cases low. On the other hand, abstract-domain optimizations, such as decomposition, require enough variables to obtain non-trivial partitions [46]. In our implementation, the number of variables is configurable; we use eight variables in our experiments.

Expressions. Testing assignments requires numerical expressions. For the numerical domains considered in this paper, these expressions are linear sums over the program variables with integer coefficients, which are chosen by the fuzzer to increase the likelihood of constructing suitable expressions. Note that, to test precision properties, we also need to generate expressions that are not representable in the domain under test. For instance, for an octagon

Table 1: Structure of abstract elements with one constraint for commonly used numerical domains.

Domain	Element with one constraint
Intervals	$\{k_l \le v_i \le k_u\}$
Zonotopes	$\{v_i = \frac{k_l+k_u}{2} + \frac{k_u-k_l}{2} * \varepsilon_i\}$
Octagons	$\{c_i v_i + c_j v_j \odot k\}$
Polyhedra	$\{c_0 v_0 + c_1 v_1 + \ldots c_{dim-1} v_{dim-1} \odot k\}$

v: variables, c: coefficients, $\varepsilon \in [-1, 1]$, k: constants, $\odot \in \{\ge, =\}$

$\{v_0 \ge 0 \wedge v_1 \ge 0\}$, the assignment $v_2 := v_0 + 2v_1$ is expected to produce $\{v_0 \ge 0 \wedge v_1 \ge 0 \wedge v_2 \ge 0\}$ (and not $\top$), even though the assigned expression is not octagonal. For the *cond* transformer, we obtain boolean expressions by comparing the linear sums to zero.

Domain elements. As explained in Sect. 3, we create a pool of abstract-domain elements (such as intervals or octagons) in two steps: Step 1 populates the pool by constructing elements using a combination of boundary and random testing, whereas step 2 expands it by applying domain operations to existing pool elements.

Besides $\top$ and $\bot$, step 1 also creates simple domain elements that contain only one constraint on the pre-defined program variables. More complex domain elements are constructed in step 2. Tab. 1 shows the structure of simple domain elements for common numerical domains. These elements can be constructed by choosing values for the constants k (e.g., the bounds of an interval) and the coefficients c. By default, we pick them from a small set of *pre-defined values* (e.g., boundary values such as 0 or ∞) and a small set of *arbitrary values*. The use of pre-defined values is optional and can be configured in the test drivers (see Sect. 5).

These sets of values depend on both the domain under test and its implementation. For instance, octagonal coefficients must be in $\{-1, 0, 1\}$, whereas polyhedral ones are arbitrary integers. Moreover, different implementations represent numbers differently; e.g., we use the pre-defined values $\{INT\_MIN, 0, INT\_MAX\}$ for integer intervals if the implementation uses machine integers, $\{-\infty, 0, \infty\}$ for arbitrary-precision integers, and additionally *NaN* for floating-point representations. Even though we focus on integer program variables here, internal floating-point computations may lead to rounding errors, as we observed in our experiments (see Sect. 5).

The values in both sets are *not* chosen by the fuzzer. However, the fuzzer can still control the pool of domain elements by selecting suitable operations in step 2. This step constructs more diverse domain elements, usually with more complex constraints, by applying domain operations to the existing elements. Step 2 makes use of all domain operators that yield domain elements ($\sqcup$, $\sqcap$, $\triangledown$) as well as the abstract transformers *assign* and *project*. For simplicity, we omit narrowing, which is not supported by all domains, and conditionals. Using all these domain operations not only allows us to detect errors in their implementation (such as the missing closure in assignments that we discussed in Sect. 3), but it also efficiently generates a diverse set of *valid* domain elements. While it is theoretically possible to create a new element by generating an arbitrary set of constraints, such an approach would often produce unsatisfiable conjunctions of constraints, represented by the already considered $\bot$ element.

5 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our approach, we apply it to two complex libraries for numerical analysis, namely APRON [31] and ELINA [46]. We were able to find errors in several of the implemented domains. Most of them are already confirmed and fixed.

APRON is a mature library, extensively used in many academic and industrial static analyzers, such as Astrée [12] and PAGAI [30], as well as in the CPAchecker verification platform [11]. ELINA is a recent library that uses highly-optimized algorithms based on online decomposition to achieve significant speedups [46, 47]. These algorithms are difficult to implement correctly.

In our experiments, we consider three variants of APRON with different internal representations for numerical values (Sects. 5.1 and 5.2), and two versions of ELINA (Sect. 5.3). We also evaluate different configurations of our technique (Sect. 5.4), and compare it to pure fuzzing and DSE (Sect. 5.5).

Experimental setup. Since the tested domains have different complexity (i.e., the implementation of Polyhedra is significantly slower), we estimated the maximum execution time required to test each property approximately one million times for Intervals, Zonotopes, and Octagons, and half a million times for Polyhedra (see Tab. 2). The values are smaller for ELINA than for APRON because ELINA's code is highly optimized. Intervals and Zonotopes were not considered for ELINA, as they were not part of the tested artifacts [3, 4]. All the experiments were performed on a 3.3 GHz Intel Xeon E5-4627 v2 CPU with 236 GB memory and RAID6 HDD.

5.1 APRON Double and Rll

APRON supports different internal representations for numerical values. For instance, the *Double* representation uses floating-point numbers, while *Rll* uses an approximation of rational numbers based on two 64-bit integers for the numerator and denominator. Compared to APRON MPQ, which uses arbitrary-precision rationals (see Sect. 5.2), these representations offer better performance, but may lose precision and cause non-termination [1]. Intervals and Octagons support Double, while Rll is available for Polyhedra.

Our experiments indeed uncovered soundness, precision, and termination problems in several domains of the latest version of APRON (0.9.10), as shown in Tab. 3. Here, the three versions of Rll refer to different test configurations that we discuss later; the domain implementation is always the same. The third column presents the total number of properties from Fig. 3 that we attempted to test for each domain. We tested only the first 41 properties for Intervals and Polyhedra since narrowing is not implemented for Intervals in APRON and not mathematically defined for Polyhedra. The reported violations in the fourth column are not necessarily all caused by different bugs. Nevertheless, observing multiple violations caused by the same bug can provide additional information

Table 2: Maximum execution time per test driver.

Domain	Maximum execution time (s)	
	APRON	ELINA
Intervals	750	not considered
Zonotopes	2'400	not considered
Octagons	1'900	700
Polyhedra	17'700	1'800

Table 3: Results for APRON Double and Rll. The third column reports how many of the properties from Fig. 3 were applicable for each tested domain. The first value in the fourth column shows the number of violated soundness properties, the second represents precision properties, and the third indicates how often errors in the domain implementation caused the test driver to crash or time out.

Variant	Domain	#Properties				Causes
		Tested	Violated			
Double	Intervals	41	0 /	0 /	0	–
Double	Octagons	46	0 /	3 /	0	rounding
Rll v1	Polyhedra	41	0 /	0 /	41	overflow
Rll v2	Polyhedra	41	5 /	15 /	21	overflow
Rll v3	Polyhedra	41	5 /	19 /	4	overflow

for error localization. We used a configuration that initializes the pool with 32 elements (step 1 of the pool population), includes *LONG_MIN* and *LONG_MAX* as boundary values, and applies 16 operations to generate more complex domain elements (step 2). This corresponds to configuration C2 from Tab. 6, which we discuss in Sect. 5.4, together with measurements on the testing time.

Intervals and Octagons. All our generic properties hold for Intervals, the simplest domain we tested. Nonetheless, we indirectly found imprecisions for $\sqsubseteq$ and $\sqcup$, by testing APRON's efficient implementation of Zonotopes [27] (discussed in Sect. 5.2), which uses the Interval operations. These issues were confirmed and fixed.

For Octagons, three precision properties (17, 22, and 23) are violated because the equality test gives imprecise results due to rounding errors. In Fig. 5, we show an example input generated by our approach that violates property 17. `oct` represents a call to the Octagons constructor. The root cause of the imprecision is the underlying double representation. For oct_3, *LONG_MAX* cannot be precisely represented as a double value. The resulting rounding error gives approximate results in the subsequent computations and makes the assertion on line 5 fail.

Polyhedra. With the same test configuration (C2), all the tests fail to terminate for Polyhedra Rll (Rll v1 in Tab. 3). The problem is that APRON enters infinite loops during step 1 of the pool construction because of unhandled arithmetic overflows. The bug can be seen when constructing at least two consecutive domain elements, as in Fig. 6. The second constructor call enters an infinite loop.

This bug causes all test drivers to time out before they even reach the test oracle. To work around it and look for additional bugs, we replaced *LONG_MIN* and *LONG_MAX* by *INT_MIN* and *INT_MAX* as pre-defined values in the pool construction. In this case (Rll v2 in Tab. 3), step 1 of the pool construction succeeds, but for 21 test drivers, step 2 times out. The remaining 20 test drivers lead to violations of soundness and precision properties. The root cause of all these failures is unhandled arithmetic overflows in various operators. Dropping pre-defined values entirely (Rll v3 in

```
1 oct1 = oct(−x0 − x5 + 1 >= 0);
2 oct2 = assign(oct1, x2, LONG_MIN);
3 oct3 = oct(x0 + LONG_MAX >= 0);
4 oct4 = meet(oct3, oct2);
5 assert meet(⊤, oct4) == oct4;
```

Figure 5: Input violating property 17 for Octagons.

```
1 poly1 = poly(−x5 − x6 + x7 >= LONG_MAX);
2 poly2 = poly(−x4 − x5 − x6 − x7 >= LONG_MIN);
```

Figure 6: Input entering an infinite loop for Polyhedra.

Tab. 3) allows us to construct the pool in all but four cases. In total, 24 soundness and precision properties fail due to overflow.

5.2 APRON MPQ

MPQ is an APRON variant that uses arbitrary-precision rationals for its internal representation. For sub-polyhedral domains (i.e., Intervals, Octagons, and Polyhedra), this variant is supposed to be sound and precise [1]. Our experiments partially confirm these theoretical guarantees: with the same setup as for APRON Double and Rll, we did not find any (generic) property violations in APRON MPQ for the three sub-polyhedral domains. However, we uncovered imprecisions for Intervals indirectly, by testing Zonotopes. Moreover, to further validate our technique, we asked three experts in abstract interpretation to insert bugs in any of the sub-polyhedral domains. Our results are presented in the following paragraphs.

Zonotopes. As opposed to sub-polyhedral domains, the structure of Zonotopes is an incomplete lattice. For this reason, not all the precision properties from Fig. 3 are expected to hold (e.g., as explained in Sect. 4.1, the least upper bound may not exist for every pair of elements). Moreover, join creates new, input-related constants [27] and thus the operator is by design non-commutative.

Initially, the pool construction step did not succeed for any of the tests due to a memory bug in the meet operator. After applying the fix, we detected additional memory exceptions, raised when creating a high number of input-related constants. Our tests also revealed imprecisions in the implementation of the equality check, meet, and project operations. Moreover, we discovered a precision bug in the partial order. Soundness property 8 uses $\sqsubseteq$ to check if the result of a join over-approximates its operands, and the bug led to a violation of this property. The developers concluded that the root cause is an imprecision in the implementation of the Intervals domain, when one of the operands of $\sqsubseteq$, $\sqcup$, or $\triangledown$ is $\bot$, represented in its canonical form through the empty interval $[1, -1]$. If $\bot$ is not handled as a special case, $[1, -1] \sqcup [-10, -5] = [-10, -1]$, for example, instead of $[-10, -5]$. This imprecision is independent of the internal representation used for numbers. Our tests for Intervals could not detect it directly because our properties are generic and do not check the precision based on the Intervals-specific definitions. All these issues were fixed by the APRON developers.

Seeded bugs in sub-polyhedral domains. We asked three abstract interpretation researchers, a post-doc and two senior PhD students with a broad experience in implementing and using various types of abstract domains and static analyses, to seed semantic bugs for our evaluation. Each expert had the task of inserting at least five soundness or precision bugs (at least one of each type) in any of the sub-polyhedral domains. We believe that the seeded bugs are representative of the kind of semantic errors that occur during the development of abstract domains.

The cumulative results are summarized in Tab. 4. For each domain, we show how many bugs were seeded, how many our technique found, and whether we observed the bugs through violations

Table 4: Results for APRON MPQ with seeded bugs. The last column shows the number of violated soundness properties, precision properties, and of crashes and assertion failures.

Domain	#Bugs		#Properties Violated		
	Seeded	Found			
Intervals	5	4	1 /	14 /	0
Octagons	6	5	2 /	15 /	5
Polyhedra	6	5	4 /	10 /	52

of soundness or precision properties, or through crashes and assertion failures in APRON's internal consistency checks. In total, we were able to find 14 out of the 17 seeded bugs. In the following paragraphs, we present two bugs that we detected and explain why three of the seeded errors could not be found.

A seeded bug in Intervals uses a slightly modified version of an unsound definition for the widening operator [38]. This definition uses $\leq$ instead of $>$ to compare two bounds, which leads to a violation of soundness property 30. Our tests reveal this problem.

A seeded bug in Octagons removes the call of closure in one special case of the assignment transformer. As explained in Sect. 3, we detect this bug by generating an assignment during step 2 of the pool construction, followed by a meet, which violates four of our precision properties because the equality check becomes imprecise.

While our approach found the vast majority of the seeded bugs, there were three it did not detect. (1) One seeded bug makes the Octagons closure less precise. Detecting this problem would require additional, octagon-specific precision properties for closure [41]. This can be easily done, but our focus here is on domain-independent properties. (2) Another seeded bug affects the precision of widening for Intervals in a way that does not violate the properties from Fig. 3. Detecting it would again require additional, domain-specific properties. (3) The last undetected bug changes the assignment operator for Polyhedra to act like *project*, making it trivially sound, but imprecise. This bug can be easily found if the input elements are polyhedra of dimension 1 (practically intervals). Our default configuration excludes such elements; adjusting the range of dimensions to include the value 1 leads to a violation of property 37, revealing the imprecision.

5.3 ELINA

To evaluate how effective our technique is on highly optimized implementations, we applied it to test ELINA. We used the same configuration as for APRON (C2) on the following abstract domains:

- *EP1*: Polyhedra with decomposition [46]
- *EOD*: Octagons with decomposition [45]
- *EO*: Octagons without decomposition [45]
- *EP2*: more recent version (including bug fixes) of *EP1*
- *EP3*: decomposed Polyhedra with further optimizations [47]

EP1 is the implementation in the artifact [3] from POPL'17 [46], and the other variants are part of the artifact [4] from POPL'18 [47]. Note that all ELINA domains are based on floating-point numbers (not on the slower arbitrary-precision rationals). This design decision may compromise precision to achieve high performance.

Initially, the pool construction for *EP1* failed for all the tests due to corner cases like $\frac{LONG\_MIN}{-1}$. This step was also not successful for *EP2* and *EP3* if the polyhedra had *LONG_MAX* coefficients (a similar issue as for APRON Double, see Sect. 5.1). To

Table 5: Results for different variants of ELINA. The first value in the third column shows the number of violated soundness properties, the second of precision properties, and the last crashes or violated assertions in ELINA's code.

Variant	#Properties Tested	Violated			Causes
EP1	41	4 /	10 /	27	overflow, ▽
EOD	41	0 /	3 /	0	rounding
EO	41	0 /	3 /	0	rounding
EP2	41	0 /	0 /	41	overflow, assertions
EP3	41	4 /	18 /	19	overflow, ▽

find additional errors, we limited the set of pre-defined values to $\{INT\_MIN, -1, 0, 1, INT\_MAX\}$. Our results are summarized in Tab. 5. Manual inspection of the failed tests revealed that most of them were caused by arithmetic overflows in different operations (e.g., *assign* and $\sqcap$). We also found issues due to overly restrictive assertions in the code as well as an incorrect implementation of widening for certain cases (e.g., widening with $\bot$). We reported these issues (and others), and they were fixed by the developer.

One of the most interesting bugs we found in *EP1* is related to an inconsistency between the two polyhedral representations. To improve performance, ELINA uses an optimized implementation of the Chernikova algorithm [49] for incremental conversion and applies all the operators on decomposed polyhedra. Such optimizations make it much more difficult to keep the two representations in sync. Our pool-construction approach was able to create polyhedra with inconsistent internal states, by applying sequences of meet and join operations that use different internal representations. As a result, the subsequent test for the soundness of *assign* failed (property 35), because the transformer returned $\bot$. The same bug was reported by another ELINA user [50] a few days before we reported it, which shows that our approach detects bugs that are relevant for users of numerical libraries. It was fixed in the meantime.

Since narrowing was not available for Octagons through ELINA's APRON interface (which we use for testing), we did not include the corresponding test drivers in this experiment (only the first 41 properties were tested, as shown in Tab. 5). For both variants of Octagons, like for APRON Double (see Sect. 5.1), properties 17, 22, and 23 were violated. These imprecisions are caused by rounding errors when performing closure and, implicitly, in the equality tests. We reported the issues and they were confirmed. However, obtaining very precise results with finite-precision representations is challenging and the developer is still working on a fix.

5.4 Different Configurations

Our technique relies on three main configurable parameters: (1) the size of the initial pool in step 1 of the input generation (see Sect. 3), (2) the number of operations applied in step 2, and (3) whether pre-defined values are used to construct elements and expressions. We now assess the impact of these parameters on its effectiveness.

For this experiment, we used the three versions of APRON MPQ with the bugs seeded by the experts and the configurations shown in Tab. 6. The results are presented in Tab. 7. For each seeded bug, we report the abstract domain in which it was inserted, the violated properties, and the execution time until each configuration detects the violation. As shown in the table, configurations C2, C5, and C6 all find the maximum number of violations, and implicitly all the bugs, within the time limits defined in Tab. 2. However, C2 does so significantly faster than C5 and slightly faster than C6.

Table 6: Different configurations of our technique.

Configuration	Initial pool size	#Operations	Pre-defined values?
C1	2	16	yes
C2	32	16	yes
C3	1024	16	yes
C4	32	0	yes
C5	32	64	yes
C6	32	16	no

Initial pool size. When the initial pool includes just $\top$ and $\bot$ (as in C1), no violations are detected for Intervals in comparison to C2, that is, 4 bugs are missed. On the other hand, a very large initial pool (C3) increases the execution time without detecting all violations. We attribute this to the fact that the size of the initial pool directly influences the number of possible arguments to the subsequent operations in step 2 and to the test oracle; exploring all of them takes more time without necessarily being more effective.

Number of operations. Without using step 2 of the pool construction (C4), some bugs and property violations are missed (see Sect. 3 for an example). However, for our technique to be effective, the number of operations should not be too large since it increases the execution time. In Tab. 7, C5 is usually slower than C2 even though both configurations find the same number of violations.

Pre-defined values. For APRON MPQ, which uses arbitrary precision rationals, both C2 and C6 find all the violations, C2 being slightly faster. Pre-defined values are particularly important for testing abstract-domain implementations based on fixed-precision numbers such as APRON Double; as our experiments show (see Sect. 5.1 and Sect. 5.3), these implementations may suffer from arithmetic overflows and rounding errors.

5.5 Fuzzing and Dynamic Symbolic Execution

In this section, we compare our technique to pure gray-box fuzzing and DSE. In particular, we use LibFuzzer [6] and KLEE [13], a state-of-the-art DSE engine, to generate inputs for the test oracles corresponding to the properties from Fig. 3. For a fair comparison, we write *alternative test drivers* that do not create and populate a pool of abstract elements. Instead, we allow the tools to directly generate the coefficients and constants of up to 50 constraints per element. The test oracles are the same as in our test drivers.

Gray-box fuzzing. We ran LibFuzzer (with default options) on the three APRON MPQ versions with seeded bugs, using the same time limits as in Tab. 2, and compared the results to our C2 configuration. LibFuzzer detected 45 out of the 58 property violations that our approach found. It revealed these violations generally faster than our approach for Intervals and Polyhedra, but slower for Octagons. A manual inspection of the generated counterexamples shows that our technique produces significantly simpler and more readable test inputs, which was very useful in debugging the detected issues.

Tab. 8 provides additional details for the bugs seeded by Expert 3. The last two columns show the time it takes to detect a property violation for our technique (with C2) and for LibFuzzer, respectively. LibFuzzer missed one of the Octagon bugs. Moreover, LibFuzzer's results for Polyhedra suggest that the implementation of join is

Table 7: Impact of different configurations on the effectiveness of our technique. The last six columns show the time needed to find a violation for each configuration from Tab. 6. We grouped the violations by the seeded bugs they reveal (dashed lines) and by the expert who seeded the bugs (solid lines). We excluded two seeded bugs in Polyhedra that caused assertions failures in APRON for all configurations, and thereby masked the other bugs seeded by Expert 2.

Domain	Violated property	Execution time (s) C1	C2	C3	C4	C5	C6
Intervals	15	ND	0.3	1.36	0.1	1.48	0.38
Intervals	17	ND	0.05	1.78	0.05	0.05	0.05
Intervals	18	ND	0.51	1.82	0.53	0.6	0.53
Intervals	19	ND	0.44	1.89	0.65	0.6	0.52
Intervals	23	ND	2.6	1.34	0.93	3.06	0.99
Intervals	24	ND	1.94	ND	ND	17.29	2.49
Intervals	25	ND	0.33	1.34	0.1	1.71	0.4
Intervals	30	ND	0.36	1.25	0.13	0.58	0.52
Octagons	6	0.04	0.05	0.49	0.1	0.06	0.06
Octagons	9	0.34	4.73	ND	0.18	30.7	3.97
Octagons	10	0.95	3.05	ND	0.13	2.81	5.12
Octagons	11	6.25	8.35	ND	0.5	294.26	2.59
Octagons	13	0.45	3.23	ND	0.22	108.69	7.67
Intervals	30	ND	0.72	1.24	0.27	2.1	0.45
Octagons	15	0.5	crash	109.56	crash	crash	3.5
Octagons	16	0.06	0.12	2.47	0.06	0.13	0.05
Octagons	17	0.72	2.72	ND	ND	crash	crash
Octagons	18	crash	crash	23.2	crash	crash	crash
Octagons	19	crash	crash	ND	crash	crash	crash
Octagons	20	0.8	crash	771.25	crash	crash	crash
Octagons	21	crash	1.09	7.49	crash	1.41	0.9
Octagons	22	0.72	crash	ND	crash	crash	crash
Octagons	23	0.73	crash	27.81	crash	crash	3.65
Octagons	24	crash	crash	ND	crash	crash	crash
Octagons	25	0.56	0.87	4.39	crash	crash	1.03
Octagons	28	crash	crash	ND	crash	crash	crash
Octagons	42	0.93	5.17	112.34	crash	crash	1.5
Octagons	38	0.05	0.07	2.04	0.05	0.07	0.14
Polyhedra	6	0.05	0.12	63.4	0.11	0.23	0.18
Polyhedra	9	1.22	30.03	ND	0.21	44.47	312.41
Polyhedra	10	0.65	2.05	ND	0.35	295.41	42.94
Polyhedra	11	54.56	419.51	ND	3.06	>4h	1487.56
Polyhedra	13	1.18	106.69	ND	0.44	483.82	86.3
Polyhedra	34	5.46	45.58	ND	0.62	33.76	85.97
Polyhedra	36	0.07	0.14	60.4	0.12	0.22	0.17
Polyhedra	37	0.48	13.0	ND	0.16	4.87	217.27
Polyhedra	38	0.46	41.75	ND	0.73	5.42	449.95
Intervals	5	ND	2.09	15.32	9.34	2.39	787
Intervals	13	ND	0.86	3.51	0.47	4.94	1.0
Intervals	23	ND	0.62	2.19	0.51	1.49	1.17
Intervals	28	ND	269	3.59	0.12	1.0	2.02
Intervals	34	ND	12.54	70.86	1.23	11.34	12.21
Intervals	39	ND	4.37	25.21	1.9	5.99	24.99
Octagons	9	0.67	0.83	3.53	0.11	0.27	0.4
Octagons	10	0.45	0.23	2.48	0.11	0.88	0.87
Octagons	13	0.65	0.55	24.58	0.32	28.67	23.94
Octagons	17	4.32	23.61	ND	ND	117.56	16.37
Octagons	22	9.78	24.22	ND	ND	17.3	11.25
Octagons	23	4.26	9.3	ND	ND	15.08	11.05
Octagons	25	3.78	11.09	ND	ND	81.92	9.17
Polyhedra	7	0.07	inc	inc	2.02	inc	inc
Polyhedra	8	0.07	0.18	64.47	0.16	0.13	inc
Polyhedra	9	0.09	0.23	66.55	0.17	0.13	inc
Polyhedra	11	inc	inc	inc	1.0	inc	inc
Polyhedra	12	0.08	0.19	65.96	0.29	0.18	0.2
Polyhedra	13	0.08	0.18	64.67	0.19	0.22	0.2
Polyhedra	25	0.08	0.2	61.76	0.19	0.14	0.19
Polyhedra	27	2.99	0.21	65.71	0.26	0.15	inc
#Violations found		**43**	**58**	**37**	**52**	**58**	**58**
#Bugs found		**8**	**12**	**11**	**11**	**12**	**12**

crash: crash due to the seeded bugs, *ND*: the violation was not detected
inc: inconsistent representations (assertion failure in APRON's code)

Table 8: Comparison of our technique with fuzzing for the bugs seeded by Expert 3. We grouped the violations by the seeded bugs they reveal (dashed lines).

Domain	Violated property	Execution time (s) Our work	Fuzzing
Intervals	5	2.09	0.39
Intervals	13	0.86	0.69
Intervals	23	0.62	0.49
Intervals	28	0.27	0.71
Intervals	34	12.54	0.39
Intervals	39	4.37	31.07
Octagons	9	0.83	7.49
Octagons	10	0.23	8.01
Octagons	13	0.55	15.16
Octagons	17	23.61	ND
Octagons	22	24.22	ND
Octagons	23	9.3	ND
Octagons	24	11.09	ND
Polyhedra	7	inc	ND
Polyhedra	8	0.18	ND
Polyhedra	9	0.23	ND
Polyhedra	11	inc	ND
Polyhedra	12	0.19	0.07
Polyhedra	13	0.18	ND
Polyhedra	25	0.2	ND
Polyhedra	27	0.21	ND

ND: the violation was not detected, *inc*: inconsistent representations (assertion failure in APRON's code)

imprecise since only property 12 is violated. In contrast, our technique revealed that the expert seeded a more serious soundness bug (properties 7, 8, and 9 are also violated).

Dynamic symbolic execution. Since KLEE does not try to explore all execution paths in external libraries and APRON makes heavy use of libraries, we performed the comparison on the *EOD* and *EO* ELINA domains, using KLEE's default options. Our alternative test drivers use ELINA's functions directly, not its APRON interface, to avoid external library calls. With the latest version of KLEE (1.4.0), all but 1 test throw an error for both tested domains, because KLEE is not able to model malloc instructions with symbolic sizes [5].

To overcome this limitation, we extended KLEE with an option for specifying the upper bound for the symbolic size; we used 8192 in our experiments. For each tested domain, our technique detected 3 violated properties (see Tab. 5), but KLEE was not able to detect any, even with a time limit of 17'700s (the 25-fold of the time limit used for our technique). We believe this is due to the fact that ELINA heavily relies on floating-point arithmetic, which KLEE does not handle very well.

5.6 Threats to Validity

We identified two threats to the validity of our experiments.

Test generation tool. Our comparisons to alternative approaches focus on one fuzzer and one DSE tool. Since we chose mature, state-of-the-art tools, we believe that our results are representative. Similarly, we did not use alternative gray-box fuzzers when evaluating our own approach. Since most fuzzers make no assumptions about the code under test, we do not expect to see significantly different results for other fuzzers.

Random initialization. Our pool initialization step chooses some of the coefficients and constants randomly. To ensure that our results are deterministic, all test drivers use the same, pre-defined random seed.

6 RELATED WORK

Our approach is the first to systematically test a wide range of soundness, precision, and convergence properties on complex abstract domains. It combines several existing test case generation techniques in a novel way. On the one hand, we derive executable test oracles [9] to check high-level, mathematical properties of abstract-domain implementations. On the other hand, we incorporate ideas from boundary and random testing (step 1 of the pool population) and from feedback-directed random testing [42] (step 2) to obtain inputs for the test oracles. A key difference with the latter is that, in our case, the fuzzer controls which elements are added to the pool, by providing the operations and their arguments.

Testing static analyzers with random programs. One way to test static analyzers is by randomly generating input programs [24]. This approach is particularly effective in testing the robustness of analyzers, that is, for detecting which input programs make the analyzers crash. To also test soundness properties, Cuoq et al. instrument the code of the analyzer under test with assertions about inferred values or relations between program variables. These assertions are then checked against concrete executions. In contrast, our technique generates input data systematically, and does not require any modifications to the implementation of the tested analyzers.

Analysis testing and delta debugging. Similarly to Cuoq et al.'s work, Andreasen et al. [8] compare concrete executions to abstract domain elements to detect soundness and precision problems. They use delta debugging to reduce the size of the input programs in order to report the errors concisely. In contrast, we propose a technique for automatically generating the input domain elements; our approach starts with simple elements and applies a small number of operations, generating small counterexamples by construction.

Systematically testing lattice properties. Midtgaard and Møller [40] focus on quickchecking [18] basic lattice properties of abstract interpreters. Our technique was inspired by their work; it extends the set of tested properties and, as our evaluation shows, is effective on widely-used and highly-optimized abstract domain implementations. A comprehensive experimental comparison with their tool was not possible, as the authors provide constructors and helper functions for generating ordered pairs of elements only for Intervals, but not for the complex numerical domains that we consider. Without them, many of the properties cannot be tested, as the randomly generated inputs very likely do not satisfy the preconditions. For this reason, we expect that their technique cannot significantly outperform gray-box fuzzing, which we showed to be less effective than our approach (see Tab. 8). We solve the problem of generating ordered inputs by applying domain operations to the existing pool elements (e.g., the result of a join over-approximates its operands).

Formally verified static analyzers. Interactive theorem provers such as Coq [2] have been used to verify the soundness of the *design* of static analyses (e.g., in the context of type systems [25, 44]). However, the proofs do not typically provide any guarantees about the actual *implementation* of the analyses, and, thus, could still benefit from automated testing techniques like ours. The Verasco [32] project extracts executable code from verified Coq formalizations of several abstract domains. This approach produces implementations that are correct by construction, but is not yet practical for complex, highly-optimized implementations. Recent work by Madsen and Lhoták [39] uses symbolic evaluation (based on symbolic executions and SMT solvers) to verify the correctness, i.e., safety and soundness, of abstract-domain implementations. If the problem is undecidable, it relies on a quickchecking approach inspired by [40]. Their evaluation does not consider complex numerical domains.

Unsoundness in static analyzers. Even for analyzers that are unsound by design [10, 15–17, 37], our technique is useful to detect *unintentional* sources of unsoundness and imprecision (e.g., caused by implementation errors).

Testing compilers and program analyzers. Abstract domains are one of many components that are used in modern compilers and program analyzers. Besides efforts in proving properties about such components (e.g., CompCert [36] and Verasco [32]), there is a significant body of work on using testing techniques to detect issues in compilers [34, 35, 48, 51], and recently, in DSE engines [33].

7 CONCLUSION

We have presented an automated testing technique for detecting soundness, precision, and convergence errors in abstract-domain implementations, which are crucial components of many static analyzers. We have evaluated our approach on several complex, real-world abstract domains from two widely-used libraries for numerical analysis and demonstrated its effectiveness in finding both seeded and previously unknown errors.

Even though our evaluation focuses on numerical abstract domains, we believe that the high-level ideas of our technique also apply to other domains such as string or heap domains. Such domains require different techniques to construct domain elements as well as suitable parameters for assignments and conditionals. Moreover, one needs to assess whether fuzzers can effectively explore the search space of non-numerical domain implementations. We leave these generalizations as future work.

There are several lessons to be learned from this work. First, an automated testing technique offers a pragmatic and effective solution for uncovering issues in static analyzers that would be difficult to find using manual testing. Second, off-the-shelf testing tools are less effective for complex, highly-optimized domain implementations than a well-designed combination of techniques. We believe that these observations carry over to other application areas, such as machine-learning frameworks; exploring those is future work.

ACKNOWLEDGMENTS

We would like to thank the developers of APRON, Khalil Ghorbal and Antoine Miné, and of ELINA, Gagandeep Singh, for their help and support. We are also grateful to Jérôme Dohrau, Gagandeep Singh, and Caterina Urban for seeding bugs for our evaluation, and to our anonymous reviewers for their helpful comments. Maria Christakis's work was supported in part by a Facebook Faculty Research Award.

REFERENCES

[1] [n. d.]. The APRON Library Documentation. http://apron.cri.ensmp.fr/library/0.9.10/apron.pdf.
[2] [n. d.]. The Coq Proof Assistant. https://coq.inria.fr.
[3] [n. d.]. ELINA Artifact (POPL 2017). https://www.sri.inf.ethz.ch/optpoly.php.
[4] [n. d.]. ELINA Artifact (POPL 2018). https://www.sri.inf.ethz.ch/popl18-paper251.php.
[5] [n. d.]. KLEE Tutorial. http://klee.github.io/tutorials/testing-regex/.
[6] [n. d.]. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.
[7] [n. d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[8] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *SOAP*. ACM, 31–36.
[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41, 5 (2015), 507–525.
[10] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *FSE*. ACM, 57–67.
[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV (LNCS)*, Vol. 6806. Springer, 184–190.
[12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *PLDI*. ACM, 196–207.
[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.
[14] Cristian Cadar and Dawson R. Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN (LNCS)*, Vol. 3639. Springer, 2–23.
[15] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *FM (LNCS)*, Vol. 7436. Springer, 132–146.
[16] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2015. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *VMCAI (LNCS)*, Vol. 8931. Springer, 336–354.
[17] Maria Christakis and Valentin Wüstholz. 2016. Bounded Abstract Interpretation. In *SAS (LNCS)*, Vol. 9837. Springer, 105–125.
[18] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. ACM, 268–279.
[19] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *ISOP*. Dunod, 106–130.
[20] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.
[21] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM, 269–282.
[22] Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP (LNCS)*, Vol. 631. Springer, 269–295.
[23] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM, 84–96.
[24] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NFM (LNCS)*, Vol. 7226. Springer, 120–125.
[25] Catherine Dubois. 2000. Proving ML Type Soundness Within Coq. In *TPHOLs (LNCS)*, Vol. 1869. Springer, 126–144.
[26] Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOOS (LNCS)*, Vol. 6528. Springer, 10–30.
[27] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The Zonotope Abstract Domain Taylor1+. In *CAV (LNCS)*, Vol. 5643. Springer, 627–633.
[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*. ACM, 213–223.
[29] Eric Goubault and Sylvie Putot. 2006. Static Analysis of Numerical Algorithms. In *SAS (LNCS)*, Vol. 4134. Springer, 18–34.
[30] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. *Electr. Notes Theor. Comput. Sci.* 289 (2012), 15–25.
[31] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV (LNCS)*, Vol. 5643. Springer, 661–667.
[32] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *POPL*. ACM, 247–259.
[33] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *ASE*. IEEE Computer Society, 590–600.
[34] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. ACM, 216–226.
[35] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*. ACM, 386–399.
[36] Xavier Leroy. 2009. Formal verification of a realistic compiler. *CACM* 52, 7 (2009), 107–115.
[37] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *CACM* 58 (2015), 44–46. Issue 2.
[38] Francesco Logozzo and Manuel Fähndrich. 2010. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* 75, 9 (2010), 796–807.
[39] Magnus Madsen and Ondrej Lhoták. 2018. Safe and Sound Program Analysis with FLIX. In *ISSTA*. ACM. To appear.
[40] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test. Verif. Reliab.* 27, 6 (2017).
[41] Antoine Miné. 2006. The Octagon Abstract Domain. *Higher Order Symbol. Comput.* 19, 1 (2006), 31–100.
[42] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 75–84.
[43] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *VMCAI (LNCS)*, Vol. 2937. Springer, 252–266.
[44] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002. A type system for certified binaries. In *POPL*. ACM, 217–232.
[45] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2015. Making Numerical Program Analysis Fast. In *PLDI*. ACM, 303–313.
[46] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *POPL*. ACM, 46–59.
[47] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. A Practical Construction for Decomposing Numerical Abstract Domains. *PACMPL* 2, POPL (2018), 55:1–55:28.
[48] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *ICSE*. ACM, 203–213.
[49] H. Le Verge. 1992. *A note on Chernikova's Algorithm*. Technical Report RR-1662. INRIA.
[50] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *ESOP (LNCS)*, Vol. 10801. Springer, 653–682.
[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. ACM, 283–294.