Differentially Testing Soundness and Precision of Program Analyzers

Christian Klinger
Saarland University, Germany
io@klinch.de

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

Valentin Wüstholz
ConsenSys Diligence, Germany
valentin.wustholz@consensys.net

ABSTRACT

In the last decades, numerous program analyzers have been developed both in academia and industry. Despite their abundance however, there is currently no systematic way of comparing the effectiveness of different analyzers on arbitrary code. In this paper, we present the first automated technique for differentially testing soundness and precision of program analyzers. We used our technique to compare six mature, state-of-the art analyzers on tens of thousands of automatically generated benchmarks. Our technique detected soundness and precision issues in most analyzers, and we evaluated the implications of these issues to both designers and users of program analyzers.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

program analysis, differential testing, soundness, precision

ACM Reference Format:
Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330553

1 INTRODUCTION

Despite the abundance of program analyzers that have been developed both in academia and industry, there is currently no systematic way of comparing their effectiveness on arbitrary code. To compare the soundness and precision of a set of analyzers, one could try them on a number of programs to get a feel for their false positive or false negative rates. However, just classifying the generated warnings as false or true positives would require considerable human effort, let alone determining whether any bugs are missed.

Alternatively, one could rely on the outcome of software verification competitions such as SV-COMP [7], which compares program analyzers based on an established collection of verification tasks. Although verification competitions are extremely valuable, the verification tasks are rather stable. As a consequence, program analyzers can be designed to perform well in such competitions by specifically tailoring their techniques to the given benchmarks.

Our approach. In this paper, we present the first automated technique for differentially testing program analyzers on arbitrary code. Given a set of seed programs, our approach automatically generates program-analysis benchmarks and compares the soundness and precision of the analyzers on these benchmarks. As a result, the effectiveness of the different program analyzers is evaluated in a systematic and automated way, the benchmarks are less predictable than the seed programs, and the explicit checks can be parameterized to express several types of properties, for instance, numerical, non-nullness, or points-to properties.

However, as for existing differential-testing techniques, it is challenging to automatically derive the ground truth, for example, which warnings are indeed true positives or which errors are missed. We address this challenge by leveraging Engler et al.'s "bugs-as-deviant-behavior" strategy [27]. Specifically, when most program analyzers agree that a certain property does not hold, our approach detects a potential soundness issue in the deviant analyzers, which find that the property does hold. Conversely, we detect a potential precision issue when a few analyzers claim that a property does not hold, while the majority of analyzers verify the property.

The work most closely related to ours is by Kapus and Cadar, who use random program generation and differential testing to find bugs in symbolic execution engines [38]. In contrast to this work, our approach focuses on detecting soundness and precision issues in any program analyzer, potentially including a test generator based on symbolic execution. Moreover, our technique automatically generates program-analysis benchmarks from a given set of seed programs, possibly containing code that is difficult to handle by program analysis. In general, it is very challenging to randomly generate programs from scratch such that they reveal soundness and precision issues in mature analyzers, which is why our technique leverages the seed programs.

Overall, our approach is a first step toward guiding users in making informed choices when selecting a program analyzer. However, this is not to say that the best analyzer is the most sound; users have varying needs depending on how critical the correctness of their code is and where in the development cycle they are (e.g., before a code review or product release) [12]. We also expect our technique to assist designers of analyzers in detecting soundness and precision issues of their implementation, and to help enrich the collection of tasks used in verification competitions by automatically generating challenging, yet less predictable, benchmarks.

Contributions. We make the following contributions:

– We present the first automated technique for differentially testing soundness and precision of analyzers on arbitrary code.

ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330553

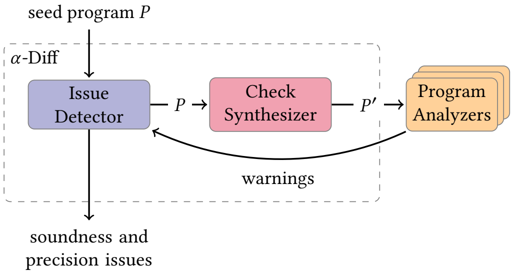

Figure 1: Overview of the workflow and tool architecture.

- We implement our technique in a tool architecture that compares analyzers on C programs and may be instantiated with any program analyzer for C.
- We report our experience and lessons learned while using this technique to compare six state-of-the-art analyzers on about 26,000 programs. We found soundness and precision issues in four (out of six) analyzers, and we evaluated their significance to both program-analysis designers and users.

Terminology. Since terminology varies across different analysis techniques, we introduce the following terms. *Sound* program analysis over-approximates the set of possible executions in a given program in order to prove the absence of errors. Due to its over-approximation, sound analysis may generate *false positives*, that is, spurious warnings about executions that are not erroneous or even possible in the program. In contrast, a *true positive* is a warning about an actual error. An *imprecise* program analysis abstracts certain program executions such that it considers more executions than those feasible in the program. Although an imprecise analysis might generate false positives, it is not necessarily sound.

2 OVERVIEW

In this section, we illustrate the workflow and tool architecture of our differential-testing technique for analyzers, shown in Fig. 1.

Workflow. Our technique, which is implemented in a tool called α-Diff, takes as input one or more correct seed programs that do not contain any (explicit or implicit) assertions. Because these programs are both correct and assertion-free, program analyzers (even if sound and imprecise) do not generate any warnings for them.

Next, α-Diff parses a seed program, and based on one of its search strategies (Sect. 3.1), a program location is selected. At this location, α-Diff synthesizes and introduces a check, in the form of an assertion, expressing a property of interest (e.g., a numerical property) and involving variables that are in scope at the location (Sect. 3.1). We call the resulting program a variant of the seed.

On this program variant, containing a single assertion, α-Diff runs a set of program analyzers and records their results, that is, the presence or absence of any generated warnings for the assertion.

Subsequently, α-Diff selects a new program location in the same seed program and repeats the process until a given *budget* (i.e., number of synthesized checks for a particular seed) is depleted. The tool then continues to parse another seed program, if any.

When all seed programs have been instrumented and analyzed until the budget, α-Diff compares the recorded results and reports any soundness and precision issues in the analyzers (Sect. 3.2).

```
1 int main() {
2   int i = 0;
3   while (i < 100000) {
4     assert i != 13; // soundness issue
5     i = i + 1;
6   }
7   assert i != 10;   // precision issue
8   return i;
9 }
```

Figure 2: Soundness and precision issues in SMACK.

Note that it is very difficult to synthesize programs such that they reveal soundness and precision issues in analyzers. For this reason, our technique takes potentially complex programs as seeds and generates variants simply by synthesizing checks. This approach has the additional benefit that any given difference in the analyzer results can only be due to a single check in a variant.

Example. Fig. 2 shows a simplified version of an SV-COMP benchmark that we used in our evaluation as a seed program to test six analyzers. (Lines 4 and 7 should be ignored for now.) Note that this program is correct and does not contain any assertions.

When passing this seed program to α-Diff, the 'Check Synthesizer' (from Fig. 1) introduces the assertion on line 4. Our tool then runs all analyzers on the resulting program variant, whose assertion can obviously fail. All analyzers detect the assertion violation except for CBMC [14], a bounded model checker for C and C++ programs, and SMACK [54], a software verifier that translates the LLVM intermediate representation into the Boogie intermediate verification language [4]. CBMC typically unrolls loops as many times as necessary such that all bugs are found, but we imposed a time limit on all analyzers, which proved to be insufficient for CBMC to unroll the loop enough times and detect the assertion violation. (Note, however, that CBMC does find the bug with a higher time limit.) Still, CBMC soundly returns 'unknown', that is, no bugs were found but, due to reaching the time limit, the program has not been verified. On the other hand, SMACK claims to have verified the assertion on line 4, which indicates a soundness issue.

We reported this issue[1] to the designers of SMACK, who told us that the assertion violation is missed due to a size-reduction heuristic, which searches for large constants in SV-COMP benchmarks, such as `100000` in Fig. 2, and replaces them with smaller numbers, in our case with `10`, to reduce the benchmark size. This heuristic is unsound and, as the SMACK designers confirmed, it is specifically tailored to the competition benchmarks.

In addition to being a source of unsoundness, this heuristic can also cause imprecision. For example, consider the program variant of Fig. 2 with the assertion on line 7 (and without the assertion on line 4). This assertion is introduced by α-Diff within the budget that is assigned to the seed program from SV-COMP, and it can never fail. All analyzers but SMACK are able to verify this program. However, because of the size-reduction heuristic, SMACK knows that variable `i` is equal to `10` right after the loop, and therefore, the verifier reports an assertion violation, indicating a precision issue.

3 DIFFERENTIAL TESTING OF ANALYZERS

We now describe the main components of our workflow in detail.

[1] https://github.com/smackers/smack/issues/324

Algorithm 1 Check synthesis

```
Input: seed program $P$
// Pick candidate expression e at program location l.
$e, l \leftarrow$ GetCandidateExpression($P$)
// Pick constant k.
$k \leftarrow$ GenerateConstant($P$, $e$)
// Create check.
$s \leftarrow$ assert $e$ != $k$
// Instrument program P by inserting statement s.
$P' \leftarrow$ InsertAtLocation($P$, $l$, $s$)
Output: program variant $P'$
```

3.1 Check Synthesis

The check-synthesis component consists of two aspects: (1) the instrumentation, which creates a check and introduces it at a certain location in the seed program to generate a variant, and (2) the search strategies, which explore the space of possible seed-program variants that may be generated. We discuss these aspects next.

Instrumentation. The checks introduced by α-Diff in a seed program target numerical properties. In particular, they check whether an expression e can ever have value k at program location l. If so, the check can fail, and the tested analyzers should detect this violation in order to be sound. If the expression can never have this value at that location, the check cannot fail, and the analyzers should not detect any violation in order to be precise. Our implementation targets such safety properties since they are found in almost every program and can, thus, be checked by most analyzers. For instance, when checking for division by zero, analyzers need to ensure that the denominator can never have the value zero; similarly, for null-pointer exceptions, etc.

Alg. 1 describes how α-Diff generates a variant P' from a seed program P. First, the algorithm selects a candidate expression e at program location l in P. In our context, a candidate expression is a pure expression of integral type that reads from at least one variable, e.g., `c + 3` or `a[i]`, where `c` is a variable of type `char` and `a` is an array of integers. (Note that the choice of expression e is made according to a search strategy.) Next, the algorithm generates a constant k, which is used, together with expression e, to create an assertion of the form `assert` e `!=` k. To generate variant P', this assertion is inserted at location l in P. As an example, consider the assertion on line 4 of Fig. 2, which is introduced at the location where expression `i` (right-hand side of the assignment on line 5) is found in seed program P. Similarly, the assertion on line 7 is inserted at the location of the `return` statement in P.

The algorithm randomly samples constants k (see Sect. 4 for more details). As a result, the generated assertions may or may not hold. The latter is useful for detecting soundness issues in the tested analyzers when an assertion violation is not detected. The former helps to identify precision issues when the assertion is not proved. The algorithm also selects a candidate expression e at a certain location. This ensures that an assertion may be added throughout the seed program to thoroughly test the analyzers—we discuss search strategies for candidate expressions below.

Note that, although α-Diff generates program variants with numerical checks, the 'Check Synthesizer' of Fig. 1 is configurable and may be extended to also synthesize other types of properties. In particular, the synthesizer could be extended for any property that may be expressed as an assertion and uses variables in scope, like the length of an existing buffer (e.g., for buffer-overflow checking) or pointers (e.g., for non-nullness checking). Still, as we discuss in Sect. 5.3, synthesizing numerical checks was sufficient for detecting soundness and precision issues in most of the analyzers we tested.

Batch checks. To reduce the number of times the analyzers are invoked, α-Diff can also synthesize assertions with multiple conjuncts, which we call 'batch checks'. For example, a variant of Fig. 2 could check whether `i` is ever equal to `10` or `11` on line 7 with the assertion `assert i != 10 && i != 11`. Recall that, at this location, SMACK knows that `i` has value `10` and would, therefore, detect a violation only due to the first conjunct of the above assertion. For such cases, α-Diff uses divide-and-conquer to eliminate conjuncts that do not cause any disagreement between the analyzers. We evaluate the effectiveness of our technique when synthesizing assertions with batch checks in Sect. 5.3.

Search strategies. In addition to generating a value k, Alg. 1 also explores the search space of possible candidate expressions e. Our technique navigates this space using a number of *static* and *dynamic* search strategies, which we evaluate in Sect. 5.3.

Static strategies. Static strategies traverse the abstract syntax tree (AST) of the seed program to collect all the candidate expressions. These strategies then compute a weight w_e for each candidate expression e (based on a weight function), sum all weights to compute the total w_t, and assign to each expression e the probability w_e/w_t of being selected by GetCandidateExpression. Overall, α-Diff supports three static strategies that differ in their weight functions:

- The *Uniform-Random* strategy selects candidate expressions uniformly. That is, all possible locations have a weight of 1.
- The *Breadth-Biased* strategy assigns to each candidate expression at location l a weight of $1/depth(l)$, where $depth(l)$ is the depth of location l in the AST. That is, larger weights are assigned to locations higher in the AST.
- The *Depth-Biased* strategy assigns to each candidate expression at location l a weight of $depth(l)$. That is, larger weights are assigned to locations lower in the AST.

Dynamic strategies. The dynamic strategies do not assign fixed weights to candidate expressions. Instead, these strategies select an initial expression and then traverse the AST in different directions to select another. Our tool supports the following two dynamic strategies that differ in how they traverse the AST:

- The *Random-Walk* strategy selects an initial candidate expression at the first possible location in the main function of the seed program. To select another expression, this strategy moves in a random direction in the AST, e.g., to the subsequent statement, the previous compound statement, or into a function call.
- The *Guided-Walk* strategy is a variation of Random-Walk. In comparison, this strategy favors moves to locations in the AST that are likely to increase differences in the running times of the analyzers, e.g., by moving deeper in a compound statement.

3.2 Detection of Unsoundness and Imprecision

A common challenge for differential-testing techniques is detecting issues with a low false-positive rate, instead of reporting all found differences. In our context, this requires determining whether the analysis results are indeed sound or precise. To address this challenge, α-Diff uses two mechanisms for detecting soundness and precision issues in the tested analyzers, namely, the deviance and unsoundness detection mechanisms.

Deviance detection. Given a program variant (with a single assertion), analyzers return one of the following verdicts: *safe* (i.e., the assertion cannot fail), *unsafe* (i.e., the assertion can fail), or *unknown* (i.e., it is unknown whether the assertion can fail, likely because the analysis times out).

The deviance-detection mechanism is inspired by Engler et al.'s "bugs as deviant behavior" [27]. Specifically, given a program variant, when the majority of analyzers return unsafe, α-Diff detects a potential soundness issue in the deviant analyzers that return safe. Conversely, α-Diff detects a potential precision issue when a few program analyzers return unsafe, while the majority of analyzers return safe.

We call an analyzer δ-*unsound* with respect to a program variant when it returns safe and δ other analyzers return unsafe. Analogously, an analyzer is δ-*imprecise* with respect to a variant when it returns unsafe and δ other analyzers return safe. For each tested analyzer, α-Diff ranks all detected soundness (resp. precision) issues in order of decreasing severity, where severity is proportional to δ, that is, to the number of disagreeing analyzers. For instance, for the program variant of Fig. 2 containing the assertion on line 7, we say that SMACK is 5-imprecise since all other analyzers disagree.

We make the following observations about the effectiveness of this mechanism. First, there need to be several program analyzers under test. For instance, if there are only two analyzers, δ may be at most one, which does not allow to effectively rank disagreements. Second, issues are missed when they affect all tested analyzers (e.g., due to common sources of unsoundness [13]). Third, the detected issues are more likely to be true positives when the value of δ is high relative to the total number of analyzers.

Regarding the last observation, note that a high δ does not necessarily mean that an issue has been found. For instance, when testing six analyzers on a safe program variant, five might be unable to prove the assertion (i.e., they return unsafe). If the sixth analyzer verifies the assertion, this mechanism would consider it 5-unsound although it is sound. In our experiments however, we did not find any such false positives for a δ greater than or equal to four.

Unsoundness detection. Certain analyzers under-approximate the set of possible executions in a program. Consequently, when such analyzers detect an error, this is inevitably a true positive (modulo bugs in the analysis itself). For instance, we consider CBMC to be an under-approximating program analyzer because it typically analyzes the program until a loop-unrolling bound is reached and uses bit-precise reasoning (i.e., no over-approximation).

When such under-approximating analyzers find that a program variant is unsafe, then the assertion in the variant can definitely fail. Therefore, any other analyzer that returns safe for the same variant is unsound, and we call it *must-unsound*. On the other hand, when an under-approximating analyzer returns safe or unknown, it is possible that it has missed an assertion violation in the variant. We, thus, do not use the results of under-approximating analyzers to draw any conclusions about imprecision in other analyzers.

This mechanism is particularly useful when the number of tested analyzers is small, unlike the deviance-detection mechanism. In particular, a single under-approximating analyzer may detect soundness issues in a single over-approximating analyzer.

4 IMPLEMENTATION

In this section, we present the details of our implementation[2].

Check synthesis. Recall from Alg. 1 that the check-synthesis component of α-Diff generates a constant k for each inserted assertion. Constants k are sampled randomly but with a few tweaks.

First, while parsing a seed program, α-Diff populates a pool of constants with any constant c encountered in the program, $c + 1$, $c - 1$, and boundary values (such as *MIN_INT*, 0, and *MAX_INT*). The constants in the pool are then used to complement the randomly sampled constants. Second, our implementation does not sample constants uniformly to avoid frequently generating very large values. Instead, α-Diff uniformly selects a bit-width for a constant and then randomly generates a sequence of bits with this width. Third, for each check to be synthesized, we use a type checker to determine the type of the candidate expression and, thus, the bit-width of the corresponding constant that will be generated (e.g., 1 bit for expressions of type `bool` and 8 bits for `char`).

Result caching. After invoking the program analyzers, our implementation persists the results of each analysis run in a database. The database is extended with caching capabilities allowing α-Diff to avoid re-running the analyzers on program variants that have been previously generated.

5 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our approach, we apply α-Diff to six state-of-the-art program analyzers. In this section, we present our experimental setup (Sect. 5.1), give an overview of the tested analyzers (Sect. 5.2), and discuss several research questions and lessons learned (Sect. 5.3).

5.1 Setup

We selected seed programs (written in C) from the SV-COMP repository of verification benchmarks [7]. We excluded all programs that contain "float", "driver", or "eca-rers2012" in their path. The first category of excluded programs cannot be handled by all program analyzers we tested (which we aimed to treat equally), while the other two categories mainly contain very large benchmarks that caused most analyzers to reach the time limit we set for our experiments. We also excluded any other benchmark that crashed our type checker, for instance, some of the programs that are automatically generated by PSYCO [30]. This left us with a total of 1,393 seed programs.

We chose these benchmarks as seed programs because five out of the six program analyzers we tested have participated in at least one SV-COMP over the years. We were, therefore, confident that the analyzers would be able to handle most of the selected programs. Moreover, SV-COMP benchmarks typically do not exhibit arithmetic overflows to avoid penalizing analyzers that are intentionally unsound with respect to overflow [13, 47].

In general, all SV-COMP benchmarks are annotated with assertions (no other crashes should be possible), and user inputs are made explicit. To use these benchmarks as seed programs, α-Diff

[2] https://github.com/Practical-Formal-Methods/adiff

had to preprocess them. First, we removed all existing assertions such that the seed programs are correct and the program analyzers do not generate any warnings for them. Second, we ran the GCC preprocessor to eliminate any macro usages and, thus, avoid any parsing issues in the analyzers.

Unless stated explicitly, we used the following default configuration of α-Diff: a time limit of 30s, 2 CPU cores, and up to 8GB of memory per analysis run, a budget of either 100 or 20% of the number of candidate expressions (whichever is smaller) per seed program, the Uniform-Random search strategy, and a batch-check size of 4. Recall from Sect. 3.1 that α-Diff can synthesize assertions with multiple conjuncts. These are called batch checks, and we refer to the number of conjuncts as the 'batch-check size'. We ran our experiments on a dual hexacore Intel® Xeon® X5650 CPU @ 2.67GHz machine with 48 GiB of memory, running Debian Stretch.

5.2 Program Analyzers

We selected the analyzers under test such that they cover a wide range of different analysis techniques. In addition, we only chose mature tools that are under active development. We give a short description of each analyzer below. Note that, unless otherwise stated, we used their default configuration.

CBMC. CBMC [14] is a bit-precise bounded model checker that unrolls loops and expresses verification conditions as SMT queries over bit-vectors. We used version `5.3` of the tool.

CPAchecker. CPAchecker [8] is a software model checker that incorporates different program-analysis techniques, such as predicate abstraction [3, 31], lazy abstraction [36], and k-induction [25]. We used the development version `51.7-svn28636` of the tool, which incorporates fixes for two soundness issues that α-Diff detected. Note that CPAchecker won the first place in SV-COMP'18 and '19.

Crab. Crab [28, 29] is an abstract interpreter that supports several abstract domains [17, 18]. Its default configuration uses the Zones domain [51], and we enabled inter-procedural analysis. The tool was built from commit `5dd7a00b5b`.

SeaHorn. SeaHorn [32] is a software model checker that expresses verification conditions as Horn-clauses and uses existing solvers to discharge them. Its default configuration uses Spacer [39, 40], which is a fork of Z3 [22] with a variant of the IC3/PDR [10] model-checking algorithm for solving verification conditions. The tool was built from commit `59c4a917a595`.

SMACK. SMACK [54] is software model checker that translates C programs to Boogie [4], which can be checked by a number of different verification back-ends. We used version `1.9` with the default configuration, which runs the Corral verifier [41]. We also enabled the `svcomp` extension and set the loop-unrolling bound to 1,000. Note that SMACK won the second place in SV-COMP'17.

Ultimate Automizer. Ultimate Automizer (or UAutomizer) is a software model checker that uses an automata-based verification approach [23, 34, 35]. We used version `0.1.23`. UAutomizer won the second and third place in SV-COMP'18 and '19, respectively.

For all program analyzers that support this, we set the machine architecture to 32-bit. We also provided a default LTL-specification file to any analyzer that requires an explicit reachability property for checking assertions.

5.3 Results

We break our experimental results down into five categories, each investigating a different research question.

RQ1: Does α-Diff detect soundness and precision issues? Given as input the 1,393 seed programs, α-Diff generated 25,960 program variants. Fig. 3 shows the number of potential soundness issues that α-Diff detected in the tested program analyzers when running them on the generated variants. Each soundness issue corresponds to a program variant that revealed an analyzer to be must- or δ-unsound. As shown in the figure, our technique detected a significant number of potential issues in four program analyzers.

We manually inspected *all* detected issues from Fig. 3 and reported unique, undocumented sources of unsoundness to the designers of Crab, SeaHorn, and SMACK. We discuss the reaction of all designers in RQ2. Note that α-Diff had previously found two soundness issues in CPAchecker, which were reported to the tool designers early on and were fixed[3,4]. We used the patched version of CPAchecker for our experiments.

As is expected, some of the issues in Fig. 3 might expose the same source of unsoundness in an analyzer. For example, assume that α-Diff generates several variants (of one or more seed programs) that use bitwise arithmetic in their assertions. For each of these variants, our tool could potentially report a soundness issue in any analyzer that does not support bit-precise reasoning. Note, however, that our variant generation does not take as input known sources of unsoundness or imprecision; this is a benefit of our technique. Of course, users may configure the check synthesis to take them into account, but we did not do that for our experiments.

As shown in the figure, CPAchecker is found to be 3-unsound for one program variant. This issue is actually a source of imprecision in three other analyzers, namely, in Crab, SeaHorn, and SMACK, and is related to bit-precise reasoning. In general, we observed that the false-positive rate of our technique depends on two main factors. First, if some of the tested program analyzers are imprecise for a given variant, α-Diff detects soundness issues in the remaining analyzers, which are, however, sound. Second, any issues that are reported for δ-unsound analyzers, where δ is small, are likely false positives. For example, when a program analyzer is 1-unsound, there exists only one disagreeing analyzer.

When inspecting the results of Fig. 3, we did not find any false positives when $\delta \geq 4$ or for must-unsound analyzers. For $\delta < 4$, we found four unique false positives due to imprecision in other analyzers. It, therefore, becomes clear that the value of δ effectively controls the false-positive rate of our technique.

Fig. 4 shows the number of precision issues that α-Diff found for the same program variants. Although the number of issues is significant, the majority of these do not correspond to bugs in the analyzers. Instead, most of the precision issues are either intended by the analysis designers (for instance, imprecise reasoning about numeric types) or inherent to the analysis (for example, imprecision in non-relational abstract domains, such as Intervals).

Overall, α-Diff found many more precision issues in Crab in comparison to the other analyzers. These issues, however, are intended since Crab favors performance over precision, similarly to several

[3] https://groups.google.com/d/msg/cpachecker-users/3JCOeNuoleA/fpr8ElaaBgAJ

[4] https://groups.google.com/d/msg/cpachecker-users/3JCOeNuoleA/YDT7LokZBwAJ

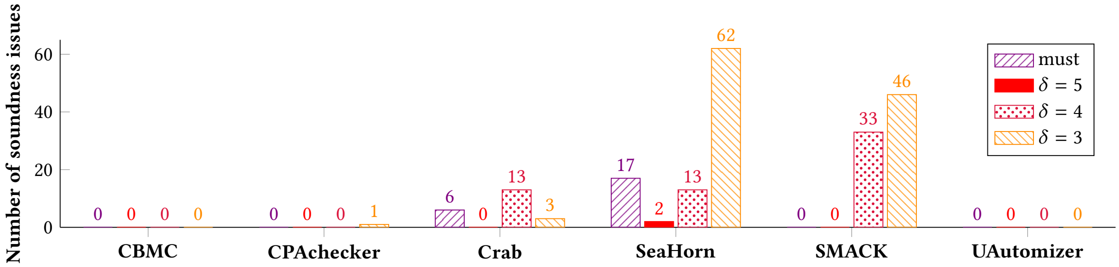

Figure 3: Soundness issues detected for each program analyzer.

abstract interpreters. Manual inspection of a random selection of these issues showed that the variants for which Crab is imprecise exhibited at least one of the following features: (1) usage of pointers, (2) bitwise operations, (3) invariants expressing parity of variables.

In addition to detecting issues in program analyzers, α-Diff can also compare their relative soundness and precision. To determine the relative precision (resp. soundness) of an analyzer A_i with respect to another analyzer A_j, our technique computes the probability that A_i returns safe (resp. unsafe) given that A_j returns safe (resp. unsafe). Tab. 1 shows these probabilities for all analyzers. Note that >0.99 stands for a probability between 0.99 and 1.00.

From the first row of the table, we observe that CBMC verifies only 1% of the variants that the other analyzers prove safe. This is due to the fact that CBMC uses bounded model checking, which might fail to explore all program paths within a certain time limit. In contrast, CPAchecker verifies 71% of the variants that CBMC proves safe. As another example, SeaHorn verifies almost all variants that UAutomizer proves, while UAutomizer verifies only 81% of the variants that SeaHorn proves. This indicates that SeaHorn is more precise on the generated variants. On the other hand, α-Diff did not detect any soundness issues for UAutomizer (Fig. 3), which could explain the higher precision of SeaHorn on some variants. In general, analyzers might gain in precision when sacrificing soundness since they consider fewer program executions.

Tab. 2 shows a direct comparison between SMACK and Crab, which implement very different analyses. Note that <1% stands for a percentage between 0 and 1. As shown in the table, for 3% of the program variants, Crab reports unsafe whereas SMACK returns safe. Both tools prove 23% of the variants safe and never return unknown for the same program variant. This suggests that these analysis techniques have complementary strengths and weaknesses.

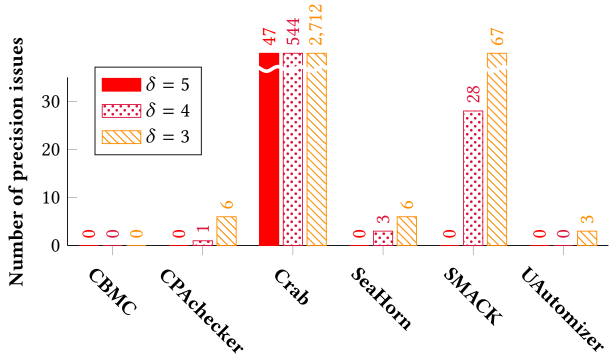

Figure 4: Precision issues detected for each analyzer.

In addition to comparing the results of different analyzers, α-Diff can also be used to compare different configurations of the same analyzer. Tab. 3 shows a direct comparison of two Crab configurations, each using a different abstract domain, namely, Octagons [52] and Polyhedra [20]. As shown in the table, there is a small number of variants that are verified with Octagons but not with Polyhedra, although in theory Polyhedra is strictly more precise than Octagons. As pointed out by the designer of Crab[5], this mismatch is due to the fact that the domains use different widening operations [16, 19] to speed up convergence of the fixed-point computation. This is a caveat [53] that was independently evaluated in a recent paper comparing different abstract domains [1].

In Tab. 4, we use α-Diff to compare the relative precision of several abstract domains of Crab, namely, Intervals, Octagons, Polyhedra, RTZ (i.e., the reduced product of disjunctive Intervals and Zones), and Zones. Across the domains, the differences in precision are small for the generated variants. However, unsurprisingly, the Intervals domain is typically less precise than the others. For instance, Intervals can only verify 89% of the variants that are proved with Zones. On the other hand, the very precise Polyhedra domain can only verify 99% of the variants that are proved with Intervals. As previously explained, this is due to the widening operation.

> Our technique is effective in detecting a significant number of soundness and precision issues in program analyzers even when slightly perturbing widely-used benchmarks.
>
> The value of δ should be high relative to the total number of tested analyzers since it effectively controls the false-positive rate of the technique.
>
> Our technique is useful in comparing the relative soundness and precision of different analyzers or of different configurations of the same analyzer.

RQ2: Are the issues relevant for analysis designers? To determine whether the issues that α-Diff reports are relevant to analysis designers, we inspected all detected soundness issues as well as a random selection of precision issues. Overall, we found ten unique, undocumented soundness and precision issues in four (out of six) program analyzers (excluding CBMC and UAutomizer). We

[5] https://github.com/seahorn/crab-llvm/issues/18

Table 1: Relative precision for the tested program analyzers.

A_j A_i	CBMC	CPAchecker	Crab	SeaHorn	SMACK	UAutomizer
CBMC	1.00	0.01	0.01	0.01	0.01	0.01
CPAchecker	0.71	1.00	0.99	0.95	0.97	0.99
Crab	0.37	0.63	1.00	0.62	0.88	0.68
SeaHorn	0.77	0.98	>0.99	1.00	0.99	>0.99
SMACK	0.72	0.53	0.76	0.53	1.00	0.61
UAutomizer	0.57	0.82	0.89	0.81	0.93	1.00

Table 2: Comparison of results for SMACK and Crab.

Crab SMACK	unsafe	unknown	safe
unsafe	14%	22%	< 1%
unknown	30%	0%	7%
safe	3%	0%	23%

Table 3: Comparison of Crab Octagons and Polyhedra.

pk oct	unsafe	unknown	safe
unsafe	43%	1%	< 1%
unknown	< 1%	10%	< 1%
safe	< 1%	< 1%	45%

reported nine of these issues to the designers of the four analyzers. All reported issues were confirmed and three (in CPAchecker and Crab) were fixed in only a few hours each. In what follows, we discuss the ten issues we found in detail. Note that we did not report documented unsoundness or imprecision in the tested analyzers, such as not detecting possible arithmetic overflows.

CPAchecker. Although for the experiments presented in this paper we used the patched version of CPAchecker, α-Diff detected two soundness issues in this analyzer, which were immediately fixed. The first issue[6] was revealed in a program variant whose simplified version is shown in Fig. 5 (line 4 should be ignored). In this program, `x` is assigned a non-deterministic integer and, therefore, the assertion on line 3 can fail. A previous version of CPAchecker missed this assertion violation due to a bug in its invariant-generation component, which was unsound when trying to obtain information about the factors of a multiplication whose product was zero.

The second issue[7] was revealed in the simplified variant of Fig. 5 when considering line 4 (and ignoring line 3). According to the C standard, the expression `x == 1` evaluates to an integer of value `0` or `1`, which is never equal to `99`. Consequently, the assertion on line 4 can fail since `x` is assigned a non-deterministic integer. CPAchecker missed this assertion violation due to a bug in its value analysis, which was unsound when analyzing nested binary expressions such as the property asserted above.

Crab. Our technique detected two soundness issues in Crab. The first issue[8] made the inter-procedural analysis of Crab unsound in the presence of recursion, and the bug was immediately fixed.

The second issue (which was reported together with the first) is caused by Crab's LLVM-based [42] front-end, which may optimize the program by exploiting undefined behavior. For example, several seed programs contain uninitialized variables. According to the C standard, the behavior of a program that reads from such variables is undefined, that is, any behavior is correct. In these cases, a compiler pass may under-approximate the behavior of the program, for instance, by assuming that any read from an uninitialized variable returns `0`, to optimize the executable code. However, this under-approximation potentially leads to unsoundness in program analyzers that analyze the optimized code. This is because the original program may fail when compiled without the optimization or with a different compiler, whereas soundness is classically defined as an over-approximation of all reachable states.

Note that our synthesized checks do not introduce any additional undefined behavior since they only compare an in-scope expression with a constant value. Therefore, any undefined behavior in the variants already exists in the seed programs themselves.

We also reported two imprecision issues[9] in the Polyhedra and Octagons domains of Crab. In particular, the less precise Intervals domain was able to verify the assertion in a program variant, whereas the more precise Polyhedra domain found the variant unsafe. As discussed in RQ1, such precision issues are possible for abstract domains with a widening operation. The issue was similar for the Octagons domain, which typically ignores dis-equalities. In contrast, given the interval $x = [0, 10]$ and the constraint $x \neq 10$, the Intervals domain does compute the more precise interval $x = [0, 9]$.

SeaHorn. We reported a soundness[10] and a precision issue[11] to the designers of SeaHorn, who confirmed both issues. The soundness issue was caused by SeaHorn's LLVM-based front-end, which is slightly different than Crab's and, thus, results in different sources of unsoundness.

Regarding the precision issue, the designers of SeaHorn explained that it is due to the conservative handling of bitwise operations and numeric types. In particular, all numeric types are abstracted into arbitrary-precision signed integers.

SMACK. We reported the soundness and precision issues[12] that are caused by the size-reduction heuristic in SMACK (see Sect. 2 for details). Although confirmed, these issues were not fixed since this behavior was intended by the designers.

We also found several other soundness issues, which were due to optimizations by SMACK's LLVM-based front-end, just like in Crab and SeaHorn. We did not report these issues to the designers since their cause is clear.

In general, the reaction of the analysis designers to all reported issues shows that α-Diff can detect important sources of unsoundness and imprecision. This is especially the case since the tested

[6] https://groups.google.com/d/msg/cpachecker-users/3JCOeNuoleA/fpr8ElaaBgAJ

[7] https://groups.google.com/d/msg/cpachecker-users/3JCOeNuoleA/YDT7LokZBwAJ

[8] https://github.com/seahorn/crab-llvm/issues/20

[9] https://github.com/seahorn/crab-llvm/issues/18

[10] https://github.com/seahorn/seahorn/issues/152

[11] https://github.com/seahorn/seahorn/issues/157

[12] https://github.com/smackers/smack/issues/324

Table 4: Relative precision for abstract domains of Crab.

D_j / D_i	int	oct	pk	rtz	zones
int	1.00	0.97	0.97	0.97	0.89
oct	0.99	1.00	0.99	0.99	0.99
pk	0.99	0.99	1.00	0.98	0.99
rtz	>0.99	>0.99	0.99	1.00	>0.99
zones	1.00	>0.99	0.99	0.99	1.00

analyzers are mature tools that are under active development. Moreover, five of these analyzers (excluding Crab) have participated in SV-COMP, which did not reveal any of the above bugs.

> Our technique detected ten unique, undocumented soundness and precision issues in four (out of six) mature program analyzers.
>
> The analysis designers confirmed all reported issues and fixed three of them, which were previously unknown.

RQ3: Are the soundness issues about undefined behavior relevant for users? Half of the tested program analyzers, namely Crab, SeaHorn, and SMACK, might be unsound in the presence of undefined behavior. As discussed earlier, this unsoundness is caused by compiler optimizations that under-approximate the possible program executions. Moreover, different compilers are inconsistent in approximating the program executions in the presence of undefined behavior and, consequently, the results of analyzers that analyze executable code can be contradictory. The above three analyzers are, therefore, sound with respect to the LLVM bit-code, but not the C source code, which users typically provide as input.

To shed more light on what users expect from program analyzers in the presence of undefined behavior, we performed a survey of 16 professional developers, who we hired on Upwork[13]. To screen the candidates, we used two short interview questions (about type-conversion rules and pointer usage in C). Out of the candidates that replied correctly, we selected those that had experience with C.

The survey contained nine short tasks. Each task included a small C program, which was a simplified version of a program variant generated by α-Diff. For every task, we asked whether the assertion in the given program can fail, and just like a program analyzer, a survey participant could respond with *yes* (i.e., unsafe), *no* (i.e., safe), and *I don't know* (i.e., unknown).

To pilot the survey tasks, we sent the survey to four students and interns who study Computer Science and already have a Bachelor's degree. We asked these participants if they found any portion of the survey difficult to understand and requested their feedback. Their responses were solely used to improve the survey.

After finalizing the tasks, we sent the survey to the professional developers. The tasks were presented to the developers in a randomized order, but in total, the survey included three unsafe programs with well-known sources of undefined behavior. The other six tasks involved programs for which most analyzers agreed regarding their safety. We used these six tasks to exclude participants who gave the wrong answer to at least four (out of six) of these questions. Based on this threshold, we excluded four (out of 16) survey participants.

```
int main() {
  int x = ★;
  assert  2 * x != 0;
  assert (x == 1) != 99 && x == 1;
  return x == 1;
}
```

Figure 5: Soundness issues in CPAchecker.

Tab. 5 shows the survey responses from the twelve developers that we did not exclude. The first column shows the task identifier: tasks 7–9 contain undefined behavior. Next to each task identifier, we indicate whether there exists an execution of the corresponding program that fails. For example, when the executable code of the programs in tasks 7–9 is not optimized, the assertions can be violated. The remaining columns of the table show the survey responses categorized as unsafe, unknown, and safe.

As shown in the table, the majority of the survey participants (ten out of twelve professional developers) considered the programs with undefined behavior to be unsafe. This suggests that program analyzers should treat undefined behavior as non-determinism, instead of optimizing it away.

On the other hand, the four excluded developers were not able to give correct answers to at least four questions from tasks 1–6, and the remaining twelve developers gave four wrong answers to these tasks (see Tab. 5). This is a strong indication that even professional C developers benefit from program analysis.

> Professional developers, who are the target users of program analyzers, consider programs that may fail due to undefined behavior as unsafe.

RQ4: What is the effect of the search strategy? To generate a seed-program variant, our technique explores the search space of all possible candidate expressions using five different search strategies (see Sect. 3.1). To evaluate how each search strategy affects the number of detected issues, we ran α-Diff on 20 seed programs, which were randomly selected from the seed programs used in the evaluation of RQ1. For each run of α-Diff on the seed programs, we enabled a different search strategy (with the same budget per seed), and we measured the number of soundness and precision issues that were found. We used the default configuration of our tool, but with a batch-check size of 1, to prevent batch checks from influencing the results.

Table 5: Survey responses from professional C developers.

Task Identifier		Survey Response		
		unsafe	unknown	safe
1	(unsafe)	11	0	1
2	(unsafe)	12	0	0
3	(unsafe)	11	0	1
4	(safe)	2	0	10
5	(safe)	0	0	12
6	(safe)	0	0	12
7	(unsafe)	10	1	1
8	(unsafe)	10	1	1
9	(unsafe)	10	1	1

[13] https://www.upwork.com/

Table 6: Effect of search strategies on the number of issues.

Search Strategy	Number of Detected Issues		
	must-unsound	≥ 3-unsound	≥ 3-imprecise
Uniform-Random	2	2	71
Breadth-Biased	1	2	70
Depth-Biased	1	1	77
Random-Walk	0	0	59
Guided-Walk	1	2	39

Tab. 6 shows the effect of the five search strategies on the number of detected issues. The first column of the table shows the search strategy, the second the cumulative number of soundness issues detected in all must-unsound analyzers, the third the soundness issues detected in ≥ 3-unsound analyzers (that is, in 3-, 4-, and 5-unsound analyzers), and the fourth the precision issues detected in ≥ 3-imprecise analyzers. In general, the results suggest that the static search strategies are more effective in detecting soundness and precision issues than the dynamic strategies. Among the static strategies, the Uniform-Random strategy helps find the most soundness issues, although the differences are small. Among the dynamic strategies, the Random-Walk strategy performs the worst. We also observed that Breadth-Biased and Guided-Walk each detect a soundness issue that is not found by any other strategy.

> The static search strategies are more effective in detecting soundness and precision issues than the dynamic strategies, with the Uniform-Random strategy finding the most soundness issues.

RQ5: What is the effect of batch checks? To evaluate the influence of the batch-check size on the effectiveness of our approach, we ran α-Diff on the same seed programs that were selected for the evaluation of RQ4. We used the Uniform-Random search strategy, and otherwise, the same configuration of our tool as in the experiment of RQ4.

Tab. 7 shows the effect of the batch-check size on the number of detected issues. Overall, larger batch-check sizes are more effective in detecting soundness and precision issues. During our experiments, we also found that larger batch-check sizes typically help in detecting the same issues faster (that is, with a smaller initial budget) in comparison to smaller sizes.

> Larger batch-check sizes are more effective and efficient in detecting soundness and precision issues.

5.4 Threats to Validity

We have identified these threats to the validity of our experiments.

Selection of seed programs. Our experimental results may not generalize to other seed programs [56]. However, we evaluated our technique by selecting seed programs from most categories of a well-established repository of verification tasks [7] and by running the program analyzers on tens of thousands of program variants. We, therefore, believe that our benchmark selection significantly helps mitigate this threat and aids generalizability of our results.

Selection of program analyzers. For our experiments, we used the program analyzers described in Sect. 5.2. Our findings depend on bugs, unsoundness, and imprecision in these analyzers and, thus, may not generalize. However, our selection includes a wide range of program-analysis techniques, like model checking and abstract interpretation. Moreover, all of these techniques are implemented in mature tools.

Table 7: Effect of batch-check size on the number of issues.

Batch-Check Size	Number of Detected Issues		
	must-unsound	≥ 3-unsound	≥ 3-imprecise
1	2	2	71
2	6	6	110
4	3	5	158
8	10	10	193
16	9	10	165
32	11	22	201

Generation of synthetic benchmarks. Synthesizing benchmarks may introduce bias. However, each variant synthesized by our technique differs from the seed program by a single assertion, which is in turn randomly generated. As a result, the perturbation of the seed is small and should not introduce systematic bias.

Type of checked properties. Our results may also not generalize to other types of checks, for example, for points-to properties. Our implementation targets numerical safety properties since they are found in almost every program and can, therefore, be checked by most analyzers. Independently, our approach and implementation are configurable and may be extended to also synthesize checks for other types of properties, for instance, pointer aliasing.

Randomness in check synthesis. Another potential threat has to do with the internal validity [56] of our experiments, which refers to whether any systematic errors are introduced in the experimental setup. A typical threat to the internal validity of experiments with randomized techniques is the selection of seeds. Recall that our check-synthesis component selects candidate expressions and constants in a randomized way. To ensure deterministic results and to avoid favoring certain program analyzers over others, α-Diff uses the same, predefined random seeds for all analyzer configurations.

Survey of developers. A potential threat to the validity of our results is that the survey questions were not understandable or clearly presented. To alleviate this concern, we piloted the survey and fine-tuned the questions based on the feedback we received. Moreover, the survey responses might not be representative of other professional C developers. However, we screened the candidates and excluded any participants who were not experienced enough.

6 RELATED WORK

In the literature, there are several techniques for evaluating different qualities of program analyzers. To ensure soundness of an analyzer, existing work has explored a wide spectrum of techniques requiring varying degrees of human effort, for instance, manual proofs (e.g., [49]), interactive and automatic proofs (e.g., [6, 9]), testing (e.g., [11, 50]), and "smoke checking" [4]. There also exist evaluations of the efficiency [57] and precision [45] of various analyses.

Our approach is the first to differentially test real-world program analyzers with the goal of detecting soundness and precision issues in arbitrary code. Specifically, we identify such issues by comparing the results of several analyzers, instead of relying on fixed test oracles. Although such a comparison requires running multiple analyzers, this is not necessarily a weakness given their abundance.

Testing analyzers on randomly generated programs. Running a program analyzer on randomly generated input programs has proved effective in revealing crashes [21]. However, it is very challenging to randomly generate programs from scratch such that they reveal soundness and precision issues in mature analyzers. Instead, our approach takes as input existing, complex programs as seeds and uses them to generate seed variants by synthesizing checks for numerical properties.

Testing symbolic execution engines. Kapus and Cadar use random program generation in combination with differential testing to find bugs in symbolic execution engines [38], by for instance comparing crashes, output differences, and code coverage. Unlike our approach, this work specifically targets symbolic execution engines and compares them on randomly generated programs.

Testing abstract interpreters. A common technique for revealing soundness issues in analyzers that infer invariants (e.g., abstract interpreters [16–18]) is to turn invariants inferred at different program locations into explicit assertions and then check if these are violated in concrete program executions [2, 21, 59]. Concrete executions (e.g., from existing test suites) are also helpful in identifying certain precision issues by observing the effect of intersecting the inferred invariants with concrete runtime values on the number of generated warnings. In contrast, our technique works for any type of safety checker. In addition, if any of the tested analyzers perform an under-approximation (e.g., a bounded model checker or a dynamic symbolic execution engine), our technique essentially compares the results of the other analyzers against a test suite that is automatically generated on the fly.

A usual source of soundness and precision issues in abstract interpreters is bugs in the implementation of the underlying abstract domains and their operations (e.g., intersection and union of abstract states). Existing techniques [11, 48, 50] for detecting such issues use well-known mathematical properties of domains as test oracles. In contrast, our approach can not only detect issues in domain implementations, but also in abstract transformers, which model program statements such as arithmetic operations or method calls. In fact, the bugs we found in Crab would not have been detected by these techniques.

Evaluating unsoundness in static analyzers. Unsoundness is ubiquitous in static analyzers [47], typically to intentionally favor other important qualities, such as precision or efficiency. A recent technique systematically documents and evaluates the sources of intentional unsoundness in a widely used, commercial static analyzer [13]. The experimental evaluation of this work sheds light on how often the unsoundness of the analyzer causes it to miss bugs. In comparison, our approach treats any tested analyzer as a black box, and it is also able to detect unintentional unsoundness and imprecision, caused by bugs in the implementation of the analyzers.

Even more recently, an empirical study evaluated how many known bugs are missed by three industrial-strength static bug detectors [33]. An important difference with our approach is that the checked properties in this study did not necessarily lie within the capabilities of the analyzers. In contrast, we synthesize numerical properties, which should be handled by all analyzers we tested. Moreover, our approach automatically synthesizes potentially erroneous programs and uses differential testing to identify both soundness and precision issues.

Formally verifying program analyzers. To avoid any soundness issues in the first place, interactive theorem provers are often used to verify the soundness of the *design* of a program analyzer. For instance, this is a common approach for type systems [26, 55]. However, such proofs cannot generally guarantee the absence of soundness issues in the actual *implementation* of the analyzer. To address this problem, the Verasco [37] project generates the executable code of several abstract domains directly from their Coq formalizations. Even if this approach were more practical, it would still not easily detect precision issues in an analyzer.

Testing compilers. Compilers typically apply different lightweight program analyses (e.g., constant propagation) to produce more efficient code. Existing work [43, 44, 46, 58, 60] has proposed several techniques for detecting bugs in compilers, and indirectly, in their analyses. These techniques often use metamorphic testing to derive test oracles [5], for instance, by comparing the output of two compiled programs where one is a slight, semantics-preserving modification of the other. In contrast, our approach compares several analyzers at once and uses their results to detect soundness and precision issues. In addition, our check synthesis instruments the seed program with assertions that may alter its semantics, for example, by introducing failing executions.

Synthetic program-analysis benchmarks. In the context of automatic test generation, LAVA [24] is a related technique that injects definite vulnerabilities, which may be triggered by an attacker-controlled input. In contrast, our instrumentation may add both assertions that do not hold to detect soundness issues as well as assertions that do hold to detect precision issues. As a result, in our case the ground truth is not known, which motivates our two mechanisms for detecting soundness and precision issues (Sect. 3.2). In addition, assertions that detect soundness issues do not necessarily need to be triggered by a program input (e.g., see Fig. 2).

7 CONCLUSION

We have proposed a novel and automated technique for differentially testing the soundness and precision of program analyzers. We used it to test six mature, state-of-the-art analyzers on tens of thousands of programs. Our technique found soundness and precision issues in four of these analyzers.

In future work, we plan to explore how to synthesize checks for different types of properties (for instance, hyperproperties [15] like information flow, and liveness properties like termination).

ACKNOWLEDGMENTS

We are very grateful to the program-analysis designers that we contacted, Matthias Dangl, Arie Gurfinkel, Jorge Navas, and Zvonimir Rakamaric, for their prompt replies and fixes as well as for their insightful explanations. We also thank the reviewers for their valuable feedback.

Maria Christakis's work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science) and a Facebook Faculty Research Award.

REFERENCES

[1] Gianluca Amato and Marco Rubino. 2018. Experimental Evaluation of Numerical Domains for Inferring Ranges. *ENTCS* 334 (2018), 3–16.

[2] 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.

[3] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *PLDI*. ACM, 203–213.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS)*, Vol. 4111. Springer, 364–387.

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41 (2015), 507–525. Issue 5.

[6] Frédéric Besson, Pierre-Emmanuel Cornilleau, and Thomas P. Jensen. 2013. Result Certification of Static Program Analysers with Automated Theorem Provers. In *VSTTE (LNCS)*, Vol. 8164. Springer, 304–325.

[7] Dirk Beyer. 2017. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org.

[8] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV (LNCS)*, Vol. 6806. Springer, 184–190.

[9] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. 2013. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *SAS (LNCS)*, Vol. 7935. Springer, 324–344.

[10] Aaron R. Bradley. 2011. SAT-Based Model Checking Without Unrolling. In *VMCAI (LNCS)*, Vol. 6538. Springer, 70–87.

[11] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.

[12] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *ASE*. ACM, 332–343.

[13] 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.

[14] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS (LNCS)*, Vol. 2988. Springer, 168–176.

[15] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *CSF*. IEEE Computer Society, 51–65.

[16] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *ISOP*. Dunod, 106–130.

[17] 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.

[18] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM, 269–282.

[19] 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.

[20] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM, 84–96.

[21] 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.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[23] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. 2017. Craig vs. Newton in Software Model Checking. In *ESEC/FSE*. ACM, 487–497.

[24] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *S&P*. IEEE Computer Society, 110–121.

[25] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software Verification Using k-Induction. In *SAS (LNCS)*, Vol. 6887. Springer, 351–368.

[26] Catherine Dubois. 2000. Proving ML Type Soundness Within Coq. In *TPHOLs (LNCS)*, Vol. 1869. Springer, 126–144.

[27] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*. ACM, 57–72.

[28] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. An Abstract Domain of Uninterpreted Functions. In *VMCAI (LNCS)*, Vol. 9583. Springer, 85–103.

[29] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. Exploiting Sparsity in Difference-Bound Matrices. In *SAS (LNCS)*, Vol. 9837. Springer, 189–211.

[30] Dimitra Giannakopoulou, Zvonimir Rakamaric, and Vishwanath Raman. 2012. Symbolic Learning of Component Interfaces. In *SAS (LNCS)*, Vol. 7460. Springer, 248–264.

[31] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (LNCS)*, Vol. 1254. Springer, 72–83.

[32] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV (LNCS)*, Vol. 9206. Springer, 343–361.

[33] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *ASE*. ACM, 317–328.

[34] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants—(Competition Contribution). In *TACAS (LNCS)*, Vol. 10806. Springer, 447–451.

[35] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *CAV (LNCS)*, Vol. 8044. Springer, 36–52.

[36] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *POPL*. ACM, 58–70.

[37] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *POPL*. ACM, 247–259.

[38] 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.

[39] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV (LNCS)*, Vol. 8559. Springer, 17–34.

[40] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV (LNCS)*, Vol. 8044. Springer, 846–862.

[41] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *CAV (LNCS)*, Vol. 7358. Springer, 427–443.

[42] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.

[43] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*. ACM, 216–226.

[44] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. ACM, 386–399.

[45] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. 2010. A Dynamic Evaluation of the Precision of Static Heap Abstractions. In *OOPSLA*. ACM, 411–427.

[46] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *PLDI*. ACM, 65–76.

[47] 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 Soundness: A Manifesto. *CACM* 58 (2015), 44–46. Issue 2.

[48] Magnus Madsen and Ondrej Lhoták. 2018. Safe and Sound Program Analysis with FLIX. In *ISSTA*. ACM, 38–48.

[49] Jan Midtgaard, Michael D. Adams, and Matthew Might. 2012. A Structural Soundness Proof for Shivers's Escape Technique: A Case for Galois Connections. In *SAS (LNCS)*, Vol. 7460. Springer, 352–369.

[50] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.* 27 (2017). Issue 6.

[51] Antoine Miné. 2004. *Weakly Relational Numerical Abstract Domains. (Domaines Numériques Abstraits Faiblement Relationnels)*. Ph.D. Dissertation. École Polytechnique, Palaiseau, France.

[52] Antoine Miné. 2006. The Octagon Abstract Domain. *HOSC* 19 (2006), 31–100. Issue 1.

[53] David Monniaux and Julien Le Guen. 2012. Stratified Static Analysis Based on Variable Dependencies. *ENTCS* 288 (2012), 61–74.

[54] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *CAV (LNCS)*, Vol. 8559. Springer, 106–113.

[55] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002. A Type System for Certified Binaries. In *POPL*. ACM, 217–232.

[56] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *ICSE*. IEEE Computer Society, 9–19.

[57] Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In *SAS (LNCS)*, Vol. 5673. Springer, 205–221.

[58] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *ICSE*. ACM, 203–213.

[59] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective Dynamic Detection of Alias Analysis Errors. In *ESEC/FSE*. ACM, 279–289.

[60] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, 283–294.