Constraint-Based Test Oracles for Program Analyzers

Markus Fleischmann
markus.fleischmann@tuwien.ac.at
TU Wien
Austria

David Kaindlstorfer
david.kaindlstorfer@tuwien.ac.at
TU Wien
Austria

Anastasia Isychev
anastasia.isychev@tuwien.ac.at
TU Wien
Austria

Valentin Wüstholz
valentin.wustholz@consensys.net
ConsenSys
Austria

Maria Christakis
maria.christakis@tuwien.ac.at
TU Wien
Austria

Abstract

Program analyzers implement complex algorithms and, as any software, can contain bugs. Bugs in their implementation may lead to analyzers being imprecise and failing to verify safe programs, i.e., programs with no reachable error locations; or worse, analyzer bugs may lead to reporting unsound results by verifying unsafe programs, i.e., programs with reachable error locations.

In this paper, we propose a method to detect such bugs by generating constraint-based test oracles for analyzers. We re-purpose and extend Fuzzle, a tool for benchmarking fuzzers, in a tool called Minotaur. Minotaur generates C programs from SMT constraints, and based on the satisfiability of the constraints, derives whether the generated programs are safe or unsafe. For instance, for an unsafe program, an analyzer under test contains a soundness issue if it proves it safe. Using Minotaur, we found 30 unique soundness and precision issues in 11 well-known analyzers that reason about reachability properties.

CCS Concepts

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

Keywords

constraint-based test oracles, program analyzers, unsoundness, imprecision

ACM Reference Format:
Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Constraint-Based Test Oracles for Program Analyzers. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695035

1 Introduction

Over the last decades, program analyzers, whether that is static analyzers or systematic-testing engines, are increasingly integrated in the software-development process. This is especially true for safety-critical code. Nevertheless, program analyzers constitute software themselves, and as a matter of fact, implement complex algorithms. They are, therefore, likely to also contain bugs in their implementations.

Such bugs can, for instance, cause a program analyzer to crash, or more importantly, return an incorrect result. In particular, the latter bugs could lead to the analyzer verifying unsafe code—i.e., having a soundness issue and producing false negatives by failing to report reachable error locations—or not being able to verify safe code—i.e., having a precision issue and producing false positives by reporting unreachable error locations as reachable. Of course, soundness and precision issues may also be intentional to favor performance, scalability, or other attributes of the analysis [9].

State of the art. Recently, there have emerged several techniques for testing program analyzers for bugs, including unintentional soundness and precision issues. Most of these techniques may be classified as applications of more general approaches, namely, *specification-based* (e.g., [7, 32, 37]), *differential* (e.g., [12, 15, 25, 26, 34, 38]), and *metamorphic testing* (e.g., [19, 29–31, 33, 39, 41, 42]). However, each of the aforementioned approaches comes with different challenges.

For example, specification-based testing requires manually and precisely formalizing the correct behavior of an analyzer, which can be difficult and impractical. While differential testing does not need manual oracle specifications, it requires multiple analyzers with the same input/output behavior in order to be applicable. Even then, the oracle is weak—it can establish that there exist one or more analyzers that behave differently but cannot determine which analyzer returns the incorrect result. Metamorphic testing lies in the middle of this spectrum: it can be used to test a single analyzer, and it requires metamorphic transformations, which are typically more lightweight than a full-blown oracle specification. Nevertheless, the metamorphic transformations need to be carefully designed in order to be applicable to code that the analyzer can handle and effective in detecting bugs.

Our technique. In this paper, we consider another general approach for testing analyzers, namely, *program generation* (e.g., [6, 22]). Specifically, this approach aims to generate programs that are safe or unsafe by construction. If running an analyzer under test on such a generated program returns a result that contradicts the ground truth about the program's safety, then a soundness or precision issue is detected. Program generation does not require any manual definitions and is applicable even for a single analyzer.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695035

However, since determining the safety of any program is an undecidable problem, this approach imposes restrictions on the generated code [22], e.g., by disallowing certain language constructs.

In 2022, Lee et al. proposed Fuzzle [27], an algorithm and tool that automatically generates buggy C programs for benchmarking fuzzers. The key idea behind Fuzzle is to cast the problem of generating unsafe programs as a maze-generation problem. In particular, each generated program resembles an agent looking for the exit in a maze. The agent consumes input that determines its sequence of movements in the maze. Once it consumes all input, the program terminates. If it finds the exit, the program aborts, i.e., a bug is found. Each maze has a single exit guaranteeing that, if the exit of the maze is reachable from the entry, the program is unsafe. To make the generated programs more realistic, Fuzzle may first use symbolic execution to compute path constraints to a number of known CVEs. It can subsequently construct program paths from these constraints leading to the maze exit. As a result, detecting the bug in a generated program may approximate the difficulty of detecting a known CVE.

Here, we describe a novel testing technique that incorporates Fuzzle for *testing program analyzers*, rather than for benchmarking fuzzers. Specifically, our technique automatically generates safe or unsafe, maze-like C programs as input to program analyzers that reason about reachability properties. It is designed based on the following insights. First, any constraint can be used to restrict the path to the maze exit in a generated program. Second, the more diverse the constraints are, the more diverse the generated programs will be, and the more thoroughly the analyzers will be tested. Third, the ground truth (i.e., safe or unsafe) for each program can be derived from the satisfiability of these constraints. This enables determining whether a detected analyzer issue is related to soundness or precision. A key challenge we faced was translating constraints, in the form of SMT formulas, to C code in a way that closely approximates their semantics.

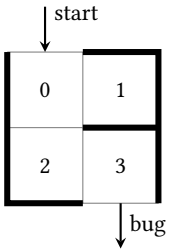
Contributions. Our paper makes the following contributions:

(1) We present a novel technique that generates safe or unsafe, maze-like C programs for testing program analyzers that reason about reachability properties.
(2) We implemented our technique in Minotaur and experimentally demonstrate Minotaur's effectiveness by testing 11 well-known program analyzers and detecting 30 soundness and precision issues.
(3) We discuss important take-aways from designing and evaluating Minotaur.
(4) We provide our tool and documentation at: https://github.com/Rigorous-Software-Engineering/minotaur

Outline. In Sect. 2, we give an overview of Minotaur, and in Sect. 3, we describe the technical details of its novel aspects. Sect. 4 highlights certain implementation choices of Minotaur. We present our experimental evaluation in Sect. 5, discuss take-aways and related work in Sects. 6 and 7, and conclude in Sect. 8.

2 Guided Tour

Here, we first provide the relevant background on Fuzzle. We then give an overview of Minotaur and present example soundness and precision issues it detected in different program analyzers.



(a) Maze

```
1  void func_bug(char *input, ...) { abort(); }
2  void func_0(char *input, ...){
3    [          ] // compute a variable used in conditionals
4    if [(      )] func_1(input, ...); // go to 1
5    else if [(      )] func_2(input, ...); // go to 2
6    else fatal_error("Hit the wall");
7  }
8  void func_1(char *input, ...){
9    [          ] // compute a variable used in conditionals
10   if [(      )] func_0(input, ...); // go to 0
11   else fatal_error("Hit the wall");
12 }
13 // func_2 is analogous
14 void func_3(char *input, ...){
15   [          ] // compute a variable used in conditionals
16   if [(      )] func_bug(input, ...); // exit
17   else if [(      )] func_2(input, ...); // go to 2
18   else fatal_error("Hit the wall");
19 }
20 // main is omitted
```

(b) Program

Figure 1: A maze and program generated by Fuzzle.

2.1 Background on Fuzzle

As already mentioned, Fuzzle generates buggy C programs for benchmarking fuzzers. The key insight behind Fuzzle is that finding a bug in a program resembles finding an exit in a maze. Therefore, every generated program represents a certain maze by encoding each path in the maze as a sequence of function calls. In turn, each function represents a particular cell in the maze. When execution reaches the maze exit, a bug is triggered; otherwise, execution terminates normally when it runs out of input.

For example, Fig. 1a shows a randomly generated 2x2 maze. Each maze cell is assigned a unique identifier, and in the figure, cell 0 is the entry cell and cell 3 the exit cell, leading to the bug. Even for such a small maze, there are arbitrarily many paths from the entry to the exit, e.g., $0-2-3$–bug, $0-1-0-2-3$–bug, etc.

Next, Fuzzle converts this maze into a program template, shown in Fig. 1b. A program template is an incomplete C program representing the maze—the boxes in the figure are holes that are filled later. In particular, each cell in the maze is encoded by a function definition, e.g., cell 0 is encoded by `func_0`, cell 1 by `func_1`, and so on. Each path in the maze is encoded by a sequence of function calls, e.g., path $0-1-0-2-3$ is encoded by calling `func_0`, satisfying the condition on line 4 to call `func_1`, satisfying the condition on line 10 to call `func_0`, satisfying the condition on line 5 to call `func_2`, and so on. The maze exit is encoded by function `func_bug` (defined on line 1 and called on line 16), which aborts execution to signal that

a bug is found. The `main` function, which is omitted in the figure, calls the maze-entry function, i.e., `func_0`, with the input.

In the final step, Fuzzle fills the holes in the generated program. The conditionals on lines 4, 5, 10, 16, and 17 are generated to be satisfiable and ensure that the branches on lines 6, 11, and 18 are never executed—an agent should not be able to go through maze walls. The holes on lines 3, 9, and 15 serve two purposes. First, they check the input size; each function consumes a different number of input bytes, and when all input is consumed, execution terminates normally. Second, they allow assigning input bytes to variables that are then used in subsequent conditionals.

More specifically, Fuzzle first fills the holes along a path to the bug. It may do this using path constraints to known CVEs, which are obtained using symbolic execution. Given such a path constraint, Fuzzle may generate several, independent conditions such that different conditions do not share variables, and thus, input bytes. Note that all conditions along a buggy path are generated to be satisfiable. All other conditions may additionally be generated using input-range or equality checks.

2.2 Overview of Minotaur

In this paper, we re-purpose and significantly extend Fuzzle to build Minotaur, which tests program analyzers for soundness and precision issues. Minotaur focuses on testing program analyzers that reason about reachability of error locations. Reachability properties are widely supported by analyzers, and several other properties may also be reduced to reachability.

A high-level overview of the main components in Minotaur is shown in Fig. 2, where our contributions are shaded in blue. Minotaur takes a Satisfiability Modulo Theories (SMT) seed constraint as input. This is similar to Fuzzle, which may use path constraints to known CVEs when generating conditions along buggy paths. We also build on the insight that constraints can be used to generate programs. In our case, we collect seed constraints from the SMT Competition (SMT-COMP) [1] and by running KLEE [8] on reachability benchmarks from the Competition on Software Verification (SV-COMP) [3].

Next, the seed constraint is passed to a constraint fuzzer that generates formula mutants—see step (1) in the figure—thereby increasing the formula diversity beyond the collected seeds and controlling the formula size. In short, our fuzzer builds on STORM [30], which aims at testing SMT solvers. STORM, however, only mutates formulas on a propositional level and only produces satisfiable formula mutants. Our fuzzer mutates formulas at a finer granularity and uses a novel technique to also generate unsatisfiable mutants. In addition to generating formula mutants, it ensures that the satisfiability of each mutant is known—this is important to later determine whether a detected issue in a program analyzer is soundness or precision related. We describe the fuzzer in detail in Sect. 3.2.

In step (2), each formula mutant is passed to the C converter, which translates it into several C expressions. These expressions are later used to fill different holes in the generated program. Note that, as described in Sect. 2.1, this functionality already exists in Fuzzle. However, its C converter is mainly tailored to handle the six CVEs included in the Fuzzle distribution. We develop a C converter that supports formulas over bit-vectors and bit-vector arrays as well as over integer arithmetic and integer arrays. Moreover, our converter can be configured to ensure that the C expressions do not contain undefined behavior—in the presence of undefined behavior, analyzers may assume that any behavior is possible. As a result, the developers of certain analyzers do not value soundness and precision issues detected in such cases (see Sect. 6).

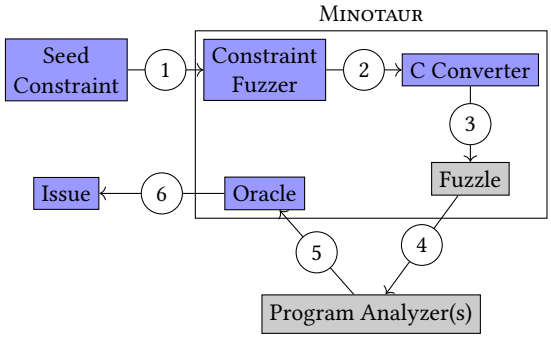


Figure 2: Schematic overview of the main components in Minotaur. Our contributions are shaded in blue.

In step (3), Fuzzle uses the C expressions to generate a maze-like program. We adapt its program generation to render the code suitable for program analysis, instead of fuzzing. In particular, we use the conventions of SV-COMP to model special information, such as non-deterministic variables (i.e., free variables in the formulas).

The generated program is then used to test one or more program analyzers, where the oracle is the satisfiability of the formula—see steps (4)–(6). Specifically, if the formula is satisfiable, we know that the bug in the program is reachable; if an analyzer under test fails to report it, a soundness issue has been detected. On the other hand, if the formula is unsatisfiable, we know that the bug in the program is unreachable; if an analyzer reports it nonetheless, a precision issue has been detected. In contrast, when using existing differential- or metamorphic-testing techniques for testing analyzers, it is unclear whether a detected issue is soundness or precision related without human inspection. When an issue is detected, Minotaur uses a minimizer to generate a smaller program that still reveals the buggy behavior of the analyzer.

2.3 Example Issues Detected by Minotaur

In the rest of this section, we showcase three example issues that were detected by Minotaur in three different program analyzers (see Fig. 3). All issues were confirmed by the analyzer developers, and the sub-figure captions link to our anonymous bug reports.

Fig. 3a shows a program that revealed a soundness issue in the invariant-generation component of CPAchecker. On line 2, variable `x` is assigned a non-deterministic value (denoted by ⋆). The analysis infers the possible values of `x` in the form of an interval, which after line 2 is [*INT_MIN*, *INT_MAX*]. The first two if-statements (on lines 3 and 4) restrict this interval to [10, 21] right after line 4. Right after line 5, the interval should be [11, 21] as, for `x = 10`, the condition of the if-statement on line 5 holds and the program returns. Instead, the interval is incorrectly inferred to be [20, 20]; this causes the analysis to unsoundly conclude that the

```
1 int main() {
2   unsigned int x = ★;
3   if (x < 10) return 0;
4   if (x > 21) return 0;
5   if (20 / x >= 2) return 0;
6   if (x < 21) return 0;
7   __VERIFIER_error();
8 }
```

(a) Soundness issue in CPAchecker.

```
1 int main() {
2   if (-((uint)-1) < 0U) {
3     __VERIFIER_error();
4   }
5   return 0;
6 }
```

(b) Precision issue in ESBMC.

```
1 ulong div(ulong l, ulong r) {
2   return l / r;
3 }
4 int main() {
5   if (div(1, (ushort)~0) == 0)
6     __VERIFIER_error();
7   return 0;
8 }
```

(c) Soundness issue in MOPSA.

Figure 3: Example soundness and precision issues found by Minotaur.

program will always return on line 6 and never reach the error on line 7.

The program in Fig. 3b triggered a precision issue in ESBMC. The program is safe, that is, the error is unreachable, because the unary minus on line 2 cannot produce a negative value. This is because the negation of unsigned integers is computed modulo the maximum integer value. However, as the expression becomes unsigned only after the cast, ESBMC fails to consider this and imprecisely claims that the error is reachable. While we reported this as a precision issue, the bug can also cause unsoundness, which is exposed by replacing `< 0U` with `> 0U` on line 2—the resulting program is unsafe, but ESBMC claims that the error cannot be reached.

Minotaur also detected a soundness issue in the MOPSA analyzer for the program in Fig. 3c. Here, the error is reachable since expression `(ushort)~0` (on line 5) yields the maximum unsigned short. As this is certainly greater than nominator `1`, the division evaluates to 0 and the condition of the if-statement holds. However, the excluded-powerset analysis in MOPSA unsoundly verifies the program. Interestingly, this soundness issue no longer occurs if function `div` is inlined.

These three issues triggered both unsoundness and imprecision in the analyzers under test and involved reasoning about non-deterministic variables as well as inter-procedural reasoning. In the following section, we discuss the technical details behind Minotaur that enable it to detect a diverse range of analyzer issues.

3 Our Approach

In this section, we describe the main components of our approach (shown in Fig. 2) in the order in which they are used.

3.1 Seed Constraint

As discussed, Minotaur uses constraints to generate more challenging programs. Based on the logics supported by the C converter, we collect constraints, in the form of SMT formulas, from SMT-COMP and by running KLEE on several SV-COMP benchmarks.

3.2 Constraint Fuzzer

As a next step, the constraint fuzzer mutates the seed constraint to obtain a configurable number of mutants of configurable size.

The constraint fuzzer serves three key functions. First, it increases the diversity of the generated programs by producing syntactically and semantically different variants of the seed constraint. Second, it regulates the size of the generated mutants, and therefore, of the generated programs. This is important as large formulas (and programs) are more likely to cause timeouts in the analyzers under test. Third, the fuzzer controls the satisfiability of the generated mutants, which in turn determines Minotaur's oracle for the analyzers under test. In other words, it determines whether a detected analyzer issue is soundness or precision related.

The fuzzer has three modes, namely, two pure modes that only generate either satisfiable or unsatisfiable mutants, and a mixed mode that generates mutants of any satisfiability. We describe them in the following.

Fuzzing SAT/UNSAT seeds to generate SAT mutants. In this mode, we fuzz any given seed formula (satisfiable or not) to generate satisfiable mutants. This is achieved by incorporating an existing fuzzer, namely STORM [30], which was developed to detect critical bugs in SMT solvers. We use those components of STORM that decompose an input formula into its propositional atoms and combine them again such that all resulting mutants are satisfiable (given a random model) and do not exceed a configurable size.

Fuzzing UNSAT seeds to generate UNSAT mutants. This mode operates on an unsatisfiable seed and implements a novel technique for generating unsatisfiable mutants. It first uses the Z3 [13] solver to extract an unsatisfiable core of the seed formula, that is, a sub-formula that is still unsatisfiable. Next, our fuzzer randomly mutates those sub-formulas in the seed that are not in the unsatisfiable core. This guarantees that the generated mutants are also unsatisfiable. We describe how the random mutations are performed later in this section.

Fuzzing SAT/UNSAT seeds to generate any mutant. Unlike the other two pure fuzzing modes, this mixed mode takes as input any seed formula (satisfiable or not) and generates mutants of any satisfiability. Our fuzzer simply creates mutants by performing random mutations on any sub-formula in the seed. Recall, however, that knowing the satisfiability of the generated mutants is important for determining the oracles for the analyzers under test and classifying detected issues as soundness or precision related. Therefore, this mode checks the satisfiability of each generated mutant post-factum, using Z3.

Random mutants. For generating unsatisfiable and mixed mutants, our fuzzer combines sub-formulas of the seed using random operators in a type-aware manner. We take inspiration from TypeFuzz [34], a fuzzer for SMT solvers that performs type-aware mutations, but unlike TypeFuzz that roughly preserves the size of the seed, our fuzzer generates mutants of a configurable size.

A seed constraint typically contains several assertions, each with a certain height of its corresponding formula tree. To generate mutants, our fuzzer takes a parameter n for the maximum number of

assertions and h for the maximum height of the formula tree that is allowed. Of course, a larger n results in more diverse mutants but is more likely to lead to the generation of contradicting assertions, thus making the mutants unsatisfiable. A larger h results in more complex mutants, and therefore, more complex programs. Given these parameters, our fuzzer generates mutants with up to n assertions as follows.

Each assertion in a mutant is constructed recursively. Given height h and a type T, which for an assertion is Boolean, the construction procedure performs two steps. First, it chooses a random operator of type T from a set of valid operators for bit-vectors, integers, and arrays, as defined in the SMT-LIB standard. For instance, operators of Boolean type are `or` and `=>` (implication), and of bit-vector type are `bvadd` (addition) and `bvor` (bitwise-or). We avoid introducing new types in the mutants by restricting the operator choice to only those where every operand type T' already appears in the seed formula. Second, the fuzzer instantiates the operands by selecting an option of type T' among the following: (1) a sub-formula of the seed with a height of no more than h; (2) a constant; or (3) a recursive call of the construction procedure with height $h-1$ and type T'.

3.3 C Converter

The C converter transforms each generated constraint into C expressions that are later used by Fuzzle to populate conditionals along the path to the maze exit. The conversion ensures that the SMT formula and the resulting C expressions are equi-satisfiable, that is, there exists a valid variable assignment for which the C expressions evaluate to a non-zero integer value if and only if the SMT formula is satisfiable.

The C converter takes as input any SMT formula in the supported logics. As mentioned earlier, the converter supports formulas over bit-vectors and bit-vector arrays as well as over integer arithmetic and integer arrays. Translation of most operations is straightforward as Boolean, arithmetic, and bitwise operations (for bit-vectors) in SMT-LIB have a unique counterpart operator in C. Certain operations are translated as a composition of C operators, for instance, implication `(=> a b)` is translated into `!a || b`, bit-vector concatenation `(bvconcat a b)` into `(a << width(a)) | b`, etc. In the following, we focus on the three more involved aspects of the conversion.

Types. Types in SMT-LIB and C do not always match, in which case we use a suitable approximation. For instance, SMT-LIB integers are unbounded, whereas C integers have finite width. We assume a 64-bit system, where integers have width 32 and long integers have width 64; we use signed long integers to represent SMT-LIB integers. (Note that we configured the tested analyzers to match this assumption.)

To ensure correct conversion, we first constrain the SMT formula further with checks expressing that all (sub-)expressions of integer type are in the range of signed long integers ($[-2^{63}, 2^{63}-1]$). We then check that the satisfiability of the resulting formula is the same as that of the original formula. For instance, if the original formula is satisfiable for unbounded integers, we check whether it is also satisfiable for signed long integers. If the check succeeds, the resulting formula sufficiently (under-)approximates the values encoded by the original formula, and the C converter proceeds with the translation into C; otherwise, the formula is disregarded.

To represent SMT-LIB bit-vectors, we use the closest unsigned integer type in C, with at least as many bits as the bit-vector. For example, a bit-vector of width 8 is translated into `unsigned char`, one of width 32 into `unsigned int`, etc. We choose unsigned types to represent bit-vectors as they have the same semantics for overflow.

However, implicit type conversions in C often require that we use explicit type casts to generate a C expression that preserves the semantics of the SMT formula. As an example, consider the following formula in SMT-LIB:

```
1 (bvsgt
2   (bvadd (_ bv255 8) (_ bv1 8))
3   (bvadd (_ bv127 8) (_ bv1 8))
4 )
```

Line 2 adds 255, represented as a bit-vector of 8 bits (`(_ bv255 8)`), and 1, also represented as a bit-vector of 8 bits (`(_ bv1 8)`). Similarly, line 3 adds 127 and 1. Then, `bvsgt` (line 1) checks whether the sum of line 2 is greater than that of line 3. Note, however, that the first sum cannot be represented in 8 bits and overflows to 0 ($11111111 + 1 = 00000000$). The second sum, on the other hand, gives $01111111 + 1 = 10000000$, which is negative in two's complement. As a result, the inequality resolves to $0 > -128$, which is true.

To preserve the SMT-LIB semantics, we translate the above formula into the following C expression:

```
1 ((signed char) ((unsigned char) (255U + 1U))
2   > -(signed char)(256U - (127U + 1U)))
```

For the first sum, we cast the result of `255U + 1U` to `unsigned char` to ensure that the overflow happens, as in SMT-LIB. Without this cast, the compiler might implicitly choose to represent the sum as an `int`. For the second sum, we explicitly compute its signed representation. Finally, we cast the left operand of the inequality to `signed char` to ensure that the compiler interprets `>` as a signed comparison. Without this cast, we would have an unsigned-character comparison giving $0 > 128$, which is false.

Arrays. By default, SMT-LIB arrays are bounded by the largest value of their index type; for instance, if the index type is `BitVec 32` (bit-vector of width 32), the array has size 2^{32}. To avoid unnecessarily large C arrays, we infer a sufficiently small array size.

For the inference, we use array reads at constant indices as a guide. In particular, we find the largest constant index appearing in the formula and initially infer the array size to be this index incremented by one. Similar to the treatment of unbounded SMT-LIB integers, we then constrain the formula further to ensure that all array accesses are within the inferred bound. Next, we check that the satisfiability of the resulting formula is the same as that of the original formula. If the original formula is satisfiable but the resulting formula becomes unsatisfiable, the inferred array size is too small to sufficiently (under-)approximate some values encoded by the original formula. We, therefore, iteratively double the inferred array size either until the resulting formula becomes satisfiable or until the inferred size exceeds 512. In the latter case, the formula is disregarded.

The array size is required for operations on an entire array, such as initialization or equality checks. Given an (inferred) array size, we

convert such operations to bounded for-loops in C. This translation naturally supports multi-dimensional arrays.

Unlike Fuzzle, our C converter supports operations at symbolic array indices. For instance, reading from index `i` of array `a`—`(select a i)` in SMT-LIB—is translated into `a[i]` in C. However, writing value `v` to index `i` of array `a`—`(store a i v)` in SMT-LIB—requires additional pre-processing.

SMT-LIB arrays are immutable, and therefore, each store operation creates and returns a new (immutable) array. To ensure correct translation into C, for each store operation, we first perform an equivalent re-write of the SMT formula as follows. We explicitly declare a new array, e.g., using a fresh symbol `FV`, add a constraint that `FV` is equal to the original array `a`, and perform the store on `FV`:

```
(and (= a FV) (...(store FV i v)...))
```

We then translate this into the following C code:

```
array_eq(a, FV) && (...array_store(FV, i, v)...)
```

Helper function `array_eq` ensures element-wise equality of the two arrays. Helper function `array_store` updates array `FV` (in place) and returns a pointer to it; this pointer is used for any subsequent read accesses.

Undefined behavior. Our C converter may be configured to generate exclusively well-defined C expressions, that is, without undefined behavior. As we discuss in Sect. 6, analyzer issues that occur in the presence of undefined behavior are less likely to be confirmed by developers, let alone fixed. We, therefore, avoid the following cases of undefined behavior.

Division by zero: SMT-LIB defines the result of division by zero for bit-vectors as a bit-vector of ones; to translate to C, we further constrain the SMT formula to check whether the denominator is zero, and if so, return the equivalent C value. For integer division, we constrain the SMT formula to ensure that the denominator is non-zero. If the resulting formula does not preserve the satisfiability of the original formula, we disregard it.

Signed overflow: We avoid signed overflow by using unsigned types wherever necessary—unsigned overflow is well defined. When casting to signed types (as for the second sum of the `bvsgt` example), we avoid any overflow by explicitly computing the corresponding signed representation.

Array accesses: We explicitly initialize array elements with non-deterministic values. We additionally check that array indices are within bounds as discussed earlier.

Bitwise shifts: To avoid shifting by more bits than the type size, we further constrain the formula to check that any expression used as the right operand of a bit-shift is within the valid range. Again, if the resulting formula does not preserve the satisfiability of the original formula, we disregard it.

3.4 Test Oracle

Given the C expressions that the converter produces from an SMT formula, Minotaur calls Fuzzle to generate a C program as described in Sect. 2.1. In short, the C expressions are used to fill in the conditions of the if-statements leading to the maze exit, and thus, the bug. Note that the exact maze and conditions are determined randomly and controlled by a random integer seed.

Minotaur uses the satisfiability of the SMT formula to derive a suitable oracle for the analyzer under test. In particular, if the formula is satisfiable, there exists a valid variable assignment for which the C expressions hold, and consequently, there exists a program trace that reaches the bug. Since the generated program is unsafe, the analyzer under test is expected to report an error. If it does not, Minotaur detects a soundness issue.

Conversely, if the formula is unsatisfiable, there is no valid variable assignment for which the C expressions hold, and consequently, there is no program trace that reaches the bug. Since the generated program is safe, the analyzer under test is expected to verify it. If it instead reports an error, Minotaur detects a precision issue.

3.5 Issue Minimization

Before reporting a detected issue, we attempt to reduce the program that revealed it.

First, our issue-minimization technique tries to reduce the SMT mutant that led to the program. To do so, it iteratively drops assertions from the formula and re-runs the rest of the Minotaur pipeline. The goal is to generate another C program, from the reduced formula, that still reveals the issue. For precision issues, we must additionally ensure that any dropped assertions are not contained in the unsatisfiable core of the formula; otherwise, the reduced formula might accidentally become satisfiable.

Second, the minimizer uses the C expression that is produced from the potentially reduced formula as the condition of a single if-statement, thereby generating a 1x1 maze. If the issue still occurs, the reduced program consists of a single function.

3.6 Discussion

Supporting other program analyses. Our approach could be used to test other program analyses as long as the properties they analyze can be expressed as reachability properties. For example, to test a taint analysis, one could add a sink in `func_bug` and the corresponding source anywhere along a path to the maze exit. To test a static call-graph analysis, one could check if `func_bug` or any other function along a path to the maze exit is transitively reachable from `main` in the call-graph. However, more elaborate encodings are also possible since they should be tailored to the analysis capabilities in order to be effective.

False positives. In general, our approach does not generate false positives. However, our implementation might do so due to bugs in Minotaur (including bugs in the underlying SMT solver). We encountered and fixed several such cases during the development of Minotaur. Nonetheless, analyzer developers might not fix issues that Minotaur detects when these are intentional sources of unsoundness or imprecision. Obviously, since our approach is based on testing, it may miss bugs in an analyzer (false negatives).

Overhead of mixed fuzzing mode. When testing analyzers that are supposed to be sound but imprecise by design, for instance, an abstract interpreter, one primarily wants to test for soundness issues. When using mixed mode in this scenario, Minotaur would have to discard every UNSAT mutant, which is inefficient. On the other hand, when testing analyzers that are supposed to be precise but unsound by design, such as a symbolic-execution engine or a bounded model checker, Minotaur would have to discard every

SAT mutant produced by the mixed mode. Consequently, discarding mutants results in performance overhead in comparison to a pure fuzzing mode.

Program diversity. The size of the programs that Minotaur generates is configurable through the maze and mutant size. Loops are introduced from cycles in the maze, and their presence is also configurable. (Note, however, that in our experiments we disallow cycles for efficiency reasons—they frequently cause timeouts in some of the tested analyzers.) The expressions Minotaur generates are as diverse as the supported SMT formulas in the logics of Tab. 1. The program templates have limited diversity since they are all instantiations of mazes.

4 Implementation

Minotaur is built on top of Fuzzle and is written in Python. To have a common interface for testing analyzers and avoid any clashes in their dependencies, each analyzer under test is built in a Docker container. In the following, we present the most interesting implementation choices we made.

Duplicate sub-formulas. SMT-COMP benchmarks and their mutants often contain duplicate sub-formulas. This may result in unnecessarily large and potentially unreadable C programs. Therefore, we (optionally) reduce the size of SMT seeds by assigning frequently occurring sub-formulas to fresh variables and replacing their occurrences with the corresponding variables. Through Minotaur's configuration, we are able to control how often this formula reduction is applied.

Expression caching. When generating mutants, certain sub-formulas may appear multiple times across mutants. To speed up the translation to C, for each sub-formula, we cache the resulting C expression. However, to ensure correct translation, the cache must be cleared whenever translation parameters change, for instance if we opt for generating only well-defined C expressions.

Parameter fuzzing. Minotaur randomizes most of its parameters as well as those of the analyzers under test. The latter is especially important; it allowed us to test various analysis components, e.g., different abstract domains in abstract interpreters, and detect issues that do not occur for the default configuration. Note that we do not blindly fuzz the analyzer configurations; we carefully chose suitable configurations for our purposes, e.g., when testing an analyzer for soundness, we only enable sound options.

5 Experimental Evaluation

We tested 11 open-source, well-known analyzers that participate in verification competitions and implement several analysis techniques, namely model checking, abstract interpretation, and symbolic execution. More specifically, we used Minotaur to test 2ls [5], CBMC [10], CPAchecker [4], ESBMC [11], MOPSA [24], SeaHorn [18], Symbiotic [35, 36], and the Ultimate framework [20], including Kojak [14], Taipan [17], Automizer [21], and GemCutter [16]. We address the following research questions:

RQ1: How effective is Minotaur in detecting soundness and precision issues in program analyzers?
RQ2: How efficient is Minotaur in detecting issues?
RQ3: How effective is Minotaur in terms of code coverage?

Table 1: Number of collected SMT seeds per logic.

Logic	BV	LIA	NIA
QF_	10358	787	50
QF_A	8320	34	8
QF_UF	157	0	0
QF_AUF	2112	36	0

5.1 RQ1: Issue Detection

Setup. For this experiment, we collected seed constraints from SMT-COMP [1]. We also generated seeds by obtaining the path conditions that KLEE computes for the ReachSafety-BitVectors and ReachSafety-Arrays categories of SV-COMP [3]. To generate these seeds, we adapted the SV-COMP programs such that they are compatible with KLEE. In the end, we excluded seeds that caused high runtime for SMT solvers, were not supported by Minotaur, or were too small to yield challenging tests (< 5 atoms). In total, we obtained almost 21,900 SMT seeds; we provide their count for each logic in Tab. 1. The first column and row show pre- and suffixes for logic names, respectively. For instance, for logic QF_BV (Quantifier-Free BitVectors), shown in the second row and column of the table, we obtained 10,358 SMT seeds.

Using these seeds, Minotaur generates mazes of size between 4x5 and 7x7 cells—the exact size is determined randomly. Generating smaller mazes was not possible due to a limitation in Fuzzle libraries, and larger mazes led to many timeouts in some analyzers under test. The algorithm for maze generation is also selected randomly (out of five possible options in Fuzzle). However, we impose the constraint that generated mazes should not contain cycles, i.e., we disallow infinite paths through the mazes. This is because cycles again led to timeouts in the tested analyzers.

We tested the analyzers for soundness issues for about 10 months, for precision issues for about 3 months, and we used Minotaur's mixed fuzzing mode for about 2 months. Note that we started testing analyzers as soon as we had a preliminary version of Minotaur. So, during the months of testing, we were also improving our tool and adding support for additional analyzers and options. In addition, we wanted to make sure that the bugs we report are unique; to do so, we typically waited for each reported bug to be fixed before testing an analyzer (configuration) further. We set the timeout of each analyzer to 60 seconds.

We ran this experiment on an AMD EPYC 7702 CPU @ 1.5GHz machine with 512GB of memory, running Debian.

Results. Tab. 2 presents all unique issues detected by Minotaur. The first column provides an identifier for each detected issue; each identifier links to the anonymized bug report. The second column lists the analyzer in which the issue was found, and the following two columns show the issue type (e.g., soundness or precision) and the fuzzing mode of Minotaur that was used (i.e., pure or mixed). Recall that 'pure' denotes fuzzing for either soundness or precision, while 'mixed' denotes fuzzing for both. The last column indicates the current status of each issue (e.g., confirmed or fixed).

Minotaur detected issues in all the 11 tested analyzers. In total, Minotaur found 31 unique issues, 19 of which are soundness related, 11 are precision related, and 1 is a crash. Of all issues, 14 are fixed, 10 are confirmed, and there is a fix planned (i.e., there is an open pull

Table 2: Issues detected by Minotaur.

Issue ID	Program Analyzer	Issue Type	Fuzzing Mode	Issue Status
1	2ls	Precision	Pure	Confirmed
2	CBMC	Precision	Mixed	Reported
3	CBMC	Precision	Mixed	Reported
4	CPAchecker	Soundness	Pure	Fixed
5	CPAchecker	Soundness	Pure	Fixed
6	CPAchecker	Soundness	Pure	Fixed
7	CPAchecker	Soundness	Pure	Confirmed
8	CPAchecker	Precision	Mixed	Fixed
9	CPAchecker	Precision	Mixed	Confirmed
10	ESBMC	Soundness	Pure	Fixed
11	ESBMC	Soundness	Pure	Fixed
12	ESBMC	Soundness	Pure	Fixed
13	ESBMC	Soundness	Pure	Fixed
14	ESBMC	Soundness	Pure	Confirmed
15	ESBMC	Precision	Mixed	Confirmed
16	MOPSA	Precision	Pure	Confirmed
17	MOPSA	Precision	Pure	Fix planned
18	MOPSA	Precision	Pure	Confirmed
19	MOPSA	Soundness	Mixed	Fixed
20	MOPSA	Soundness	Mixed	Confirmed
21	MOPSA	Soundness	Mixed	Fixed
22	MOPSA	Soundness	Mixed	Fix planned
23	SeaHorn	Soundness	Pure	Confirmed
24	SeaHorn	Precision	Pure	Reported
25	SeaHorn	Precision	Pure	Reported
26	Symbiotic	Soundness	Pure	Confirmed
27	Symbiotic	Soundness	Pure	Reported
28	Ultimate	Soundness	Pure	Fixed
29	Ultimate	Soundness	Pure	Fixed
30	Ultimate	Crash	Pure	Fixed
31	Ultimate	Soundness	Mixed	Fixed

request) for 2. The analyzer developers found our bug reports easy to investigate (see comments for issue 4) and perfectly minimal (see comments for issue 6).

Note that issues 28 and 31 are revealed for all Ultimate analyzers that we tested, issue 29 does not occur for Taipan, and issue 30 only occurs for Kojak. Moreover, issues 3, 14 and 15 were detected using our KLEE-generated SMT seeds, and all other issues using SMT-COMP seeds, which however also contain KLEE-generated formulas.

In the following, we showcase six issues detected by Minotaur and fixed by the analyzer developers (see Fig. 4). Our goal is to provide a better understanding of the kinds of issues that can be found with our technique.

Fig. 4a shows the program that revealed soundness issue 5 in CPAchecker. Variables `x` and `y` are assigned non-deterministic values (line 2), and the equality of `x` with the bitwise negation of `y` is checked on line 3. There, of course, exist values for these variables such that the equality check succeeds, and therefore, the error on line 4 is reachable. CPAchecker, however, missed this error and unsoundly verified the program. This is because its invariant-generation component incorrectly handled bitwise negation as a set-complement operation. The program in Fig. 4b revealed another soundness issue in CPAchecker (issue 6). Here, the analyzer did not soundly handle the case where the left modulo operand is 0 (line 3) and missed reporting the reachable error on line 4.

For the program in Fig. 4c, ESBMC does not report the reachable error on line 5 (issue 11) when running in interval-analysis mode. This issue occurred because the analysis removed type casts, such as the one on line 3, when processing path conditions of nested if-statements. The program in Fig. 4d revealed another soundness issue in ESBMC (issue 12). Here, the interval analysis was unsound because it was missing a case when handling Boolean negation.

The program that revealed issue 19 in MOPSA is shown in Fig. 4e. Given a non-deterministic value for variable `r`, the condition on line 4 can be satisfied. However, MOPSA unsoundly considers the error on line 6 unreachable (when using the congruence abstract domain). This issue occurred, only occasionally, when the left operand of a modulo was negative. Interestingly, for a slightly different syntax—when uncommenting line 2 and replacing line 4 with 5—this issue did not occur, and the error was soundly reported. Soundness issue 21 (not shown in the figure) was also detected in MOPSA, but the bug was actually caused by the Apron library [23], which is used by MOPSA. The bug had previously been found and fixed there, but MOPSA was using the release version of the library, which did not yet include the fix.

Minotaur detected a soundness issue in the Ultimate framework itself (i.e., affecting all four analyzers we tested) with the program of Fig. 4f (issue 28). Given non-deterministic variable `uc`, the error on line 8 is reachable regardless of type casts, e.g., for `uc = 32`. The analyzers failed to report the error due to a bug related to the right operand of a right shift being a constant.

5.2 RQ2: Performance

Setup. In this question, we evaluate the efficiency of Minotaur in detecting issues and compare it with the following two baselines:

Fuzzle: We use its default configuration, which generates all conditions using input-range or equality checks, i.e., no CVEs are used (see Sect. 2). This baseline generates programs where the error is always reachable, and therefore, it may only be used to detect soundness issues.

Fuzzle + SMT: This refers to Fuzzle extended with our C converter, but without our SMT fuzzer. Recall that Fuzzle can only handle the six CVEs included in its distribution. This baseline is able to handle all logics that Minotaur supports. The oracle is determined based on the satisfiability of the given SMT formula. In particular, for satisfiable (resp. unsatisfiable) formulas, an analyzer under test is expected (resp. not) to report the error.

We measure the performance of these baselines and Minotaur in re-finding the fixed issues of Tab. 2. We target fixed issues to be able to verify that it was indeed the target issue that was re-found, and not a different one. In particular, for each detected issue, we apply the corresponding fix implemented by the analyzer developers; if the issue no longer occurs, the detected issue is the target issue. We omit issue 30, which causes Ultimate to crash, as it is a by-product of testing for soundness and precision issues.

```c
int main() {
  int x = ★, y = ★;
  if (x == ~y) {
     __VERIFIER_error();
  }
  return 0;
}
```

(a) Soundness issue in CPAchecker.

```c
int main() {
  uint n = ★;
  if ((0 % n) <= 100) {
    __VERIFIER_error();
  }
  return 0;
}
```

(b) Soundness issue in CPAchecker.

```c
int main() {
  uint n = ★;
  if (!(0 <= (int) n))
    if (846 <= n)
      __VERIFIER_error();
  return 0;
}
```

(c) Soundness issue in ESBMC.

```c
void main() {
  bool b1 = ★, b2 = ★;
  long l1 = ★, l2 = ★;
  if (b2)
   if (!((b1 || (l2 == 0)) && !b2))
     if (!((b2 || (l1 == 0)) && !b1))
      __VERIFIER_error();
}
```

(d) Soundness issue in ESBMC.

```c
int main() {
  // signed short l = -1 % 8;
  signed short r = ★;
  if ((-1 % 8) <= r)
  // if (l <= r)
    __VERIFIER_error();
  return 0;
}
```

(e) Soundness issue in MOPSA.

```c
void main() {
  uchar uc = ★;
  if ((uint)((uint)(
        ((uint)(((uint) uc) +
          ((uint)(((uint)4294967295)*1))))
        >> ((uint) 2)))
      < (uint) 8)
    __VERIFIER_error();
}
```

(f) Soundness issue in Ultimate.

Figure 4: Issues detected by Minotaur and fixed by the analyzer developers.

For Fuzzle + SMT and Minotaur, we provide a single SMT seed, that is, the seed that originally led to detecting each issue. If we are trying to re-find a soundness issue but the SMT seed is unsatisfiable, we negate it such that Fuzzle + SMT also has a chance to detect it. We allow all techniques to generate 1000 programs per issue, and we run them 5 times, each with a different random integer seed to account for randomness in the testing processes. We again set the timeout of each analyzer to 60 seconds.

We ran this experiment on an AMD Ryzen 7 PRO 7840U CPU @ 3.3GHz machine with 32GB of memory, running Ubuntu.

Results. Tab. 3 presents the results of the comparison. The first column shows the issue identifier from Tab. 2. For each technique, we show the average time (in minutes) to re-find an issue as well as the number of different random integer seeds (out of 5) for which the issue was re-found. We use '-' and 0/5, respectively, to indicate that an issue could not be re-found within the budget of this experiment. Moreover, we use 'n/a' to express that the Fuzzle baseline cannot be used to detect precision issues (issue 8 in the table). Fuzzle + SMT is also unable to detect issue 8 as the SMT seed is satisfiable.

Table 3: Average time (in minutes) to re-find a fixed issue for Fuzzle, Fuzzle + SMT, and Minotaur, and the number of different random seeds for which the issue was re-found.

Issue	Fuzzle		Fuzzle + SMT		Minotaur	
ID	Time	Seeds	Time	Seeds	Time	Seeds
4	-	0/5	**6.5**	5/5	14.0	5/5
5	-	0/5	7.4	5/5	**4.4**	5/5
6	-	0/5	1.8	5/5	**1.4**	5/5
8	n/a	n/a	n/a	n/a	**297.1**	5/5
10	-	0/5	-	0/5	**116.2**	5/5
11	-	0/5	-	0/5	**61.4**	5/5
12	-	0/5	-	0/5	**1.1**	5/5
13	**0.9**	5/5	-	0/5	1.1	5/5
19	-	0/5	-	0/5	**5.4**	3/5
21	-	0/5	-	0/5	**6.5**	3/5
28	-	0/5	-	0/5	-	0/5
29	-	0/5	-	0/5	**13.1**	5/5
31	-	0/5	-	0/5	**213.2**	2/5

Within the budget of this experiment, Minotaur consistently (i.e., with ≥3 random seeds) detects 11 out of 13 issues, while the two baselines are much less effective. In particular, Fuzzle is only able to detect issue 13. This is to be expected given the simplicity of the generated conditions. Fuzzle + SMT only detects 3 issues. The explanation is twofold: (1) the constraint fuzzer significantly contributes to the effectiveness of Minotaur, and (2) the SMT seeds can be large and complex, thereby producing large and complex programs that frequently cause analyzer timeouts. In contrast, fuzzing the SMT seeds allows Minotaur to control the size of the mutants.

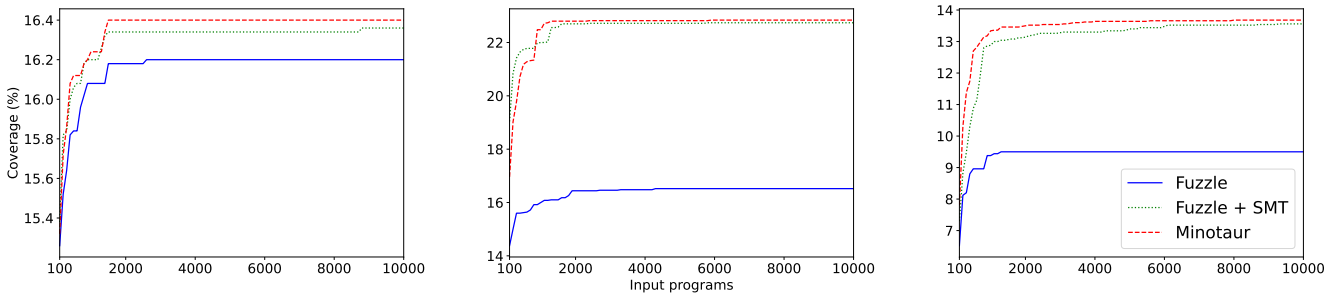
Note that, to clearly demonstrate the benefit of constraint fuzzing, we configured Minotaur to only use seed-constraint mutants for program generation, and not the seed constraint itself (as Fuzzle + SMT). This leads to Fuzzle + SMT finding 1 issue faster than Minotaur. However, Minotaur can easily be configured to subsume the Fuzzle + SMT baseline.

5.3 RQ3: Code Coverage

Setup. In this research question, we compare the coverage of analyzers under test achieved by the two baselines and Minotaur.

The analyzers we tested are written either in OCaml (MOPSA), Java (Ultimate, CPAchecker) or C (all others). We, therefore, focus on C, the most common source-code language of these analyzers. We measure the achieved coverage of CBMC, ESBMC, and SeaHorn (using gcovr [2]). We omit 2ls as it shares the code base with CBMC. We also omit Symbiotic as it combines multiple analyzers. We use branch coverage since it is the most common coverage metric for fuzzers.

We compute the average branch coverage (in %) of each analyzer across 5 randomized runs. Each run tests the analyzer on 10K



(a) CBMC coverage (b) ESBMC coverage (c) SeaHorn coverage

Figure 5: Average branch coverage (in %) of analyzer source code achieved with 10K generated programs.

programs; for Fuzzle + SMT and Minotaur, these are generated from 25 randomly selected SMT seeds. As the Fuzzle baseline is unsuitable for testing precision, we configure Fuzzle + SMT and Minotaur to only test for soundness issues, i.e., we choose the SMT seeds and mutants to be satisfiable.

We again set the analyzer timeout to 60 seconds and ran our experiments on the same hardware as for RQ1.

Results. Fig. 5 shows the average branch coverage achieved in steps of 100 programs. *Minotaur achieves the highest coverage for all three analyzers.* As expected, the Fuzzle baseline is the least effective, indicating that complex expressions derived from SMT formulas help exercise more diverse code in analyzers. When comparing Fuzzle + SMT with Minotaur, the difference in coverage is not large, yet it seems to be significant when considering the issues that Fuzzle + SMT fails to detect (Tab. 3).

5.4 Threats to Validity

Our experimental results depend on the seed constraints, the analyzers under test, and the parameters of both the analyzers and Minotaur. To mitigate these threats, we collected a large set of almost 22K SMT seeds and selected 11 program analyzers, implementing different techniques. Moreover, we randomized the parameters of both the analyzers and Minotaur during testing.

6 Discussion

In this section, we discuss insights we gained while developing and evaluating Minotaur that could benefit the community.

Detected issues. We observed that several issues were caused by the analyzers mishandling corner cases (for example, see Fig. 4a with the relatively rare bitwise-negation operator, or Fig. 4e with the modulo operator and its negative operand). Such corner cases are easy to overlook and can remain undetected by the manually crafted test suites of analyzer developers.

Issue fixes. When reporting an issue to analyzer developers, we noticed that their responsiveness and willingness to provide a fix correlate with two important factors. First, as expected, soundness issues are prioritized over precision issues. In particular, of the 19 soundness issues that we reported, 12 are fixed and one has a planned fix, whereas of the 11 precision issues that we reported, only 1 is fixed and a fix is planned for another. This is not surprising, as failing to report a reachable error location is far more critical than emitting a false alarm.

Second, bug reports containing minimal (or at least small) programs are more likely to be considered. Of course, larger programs require more debugging time and effort. Interestingly, we found that, even after minimization, programs that reveal precision issues tend to be larger than ones revealing soundness issues. This might be due to the fact that over-approximation in an analysis accumulates with more analyzed code, and it could also explain the low fix rate of precision issues.

Undefined behavior. The execution of C code with undefined behavior may be unpredictable. We noticed that handling undefined behavior in program analyzers can also be unpredictable. For example, an analyzer that reasons about reachability could soundly assume that an error location is reachable in the presence of undefined behavior, or completely ignore this possibility.

Handling undefined behavior may even vary across different analyzer configurations. For example, consider the following issue that Minotaur detected in Ultimate (not shown in Tab. 2). In its theory of bit-vectors, a division by zero yields a bit-vector with every bit set. On the other hand, in its theory of integers, a division by zero yields any integer[1]. Similarly, Minotaur found that right shifts where the left operand is negative are handled differently across the abstract domains of MOPSA (not shown in Tab. 2). While the developers originally wanted to fix the issue and consistently handle this case, they later decided against it. They mentioned that they "expect cases in which this feature is helpful to be really rare" and "probably not worth the time and the addition of complexity"[2].

When our bug reports contained programs with undefined behavior, the issues were less likely to be confirmed, let alone fixed. Soundness issues detected with such programs were sometimes fixed provided that a well-defined trace could also reach the error location. Precision issues, on the other hand, were never fixed and were not considered interesting by the developers.

Program analyzers as test subjects. During the development of Minotaur, we faced three main applicability challenges. First, while several analyzers take C programs as input, we found that they may support different language subsets and standard-library

[1]https://github.com/ultimate-pa/ultimate/issues/664#issuecomment-2097745668
[2]https://gitlab.com/mopsa/mopsa-analyzer/-/merge_requests/215#note_1921395336

features. We also observed that external library calls are rarely handled. In Minotaur, we focused on a language subset that is shared among the tested analyzers.

Second, the analysis output also varies across tools. Fuzzers like Minotaur then have to parse each individual output format separately. Any efforts to standardize the analyzer output formats (as done, for instance, in SV-COMP) greatly benefit tools and users relying on more than a single analyzer.

Third, each analyzer comes with a large variety of custom options, for instance, configuring its soundness, precision, or performance. However, these options are often poorly documented, thus making it difficult to determine whether the reported results are intentionally unsound or imprecise. Moreover, not all analyzer configurations exhibit the same soundness and precision issues; it is, therefore, critical to document differences in the expected behavior.

7 Related Work

We presented Minotaur, a technique for generating programs that are safe or unsafe by construction and using these programs to test analyzers for soundness and precision issues. In this section, we compare it with the most closely related work on analyzer testing.

Program generation. As mentioned in the introduction, we classify our technique under the general program-generation approach for testing analyzers. This approach has been applied to detect bugs in different kinds of analyzers, such as string solvers [6] and a deductive verifier [22]. For string solvers, the authors synthesize formulas that are satisfiable or unsatisfiable by construction; this ground truth is then used as a test oracle for the string solvers. In the case of the deductive verifier, the authors target Dafny [28]. They build XDsmith, which generates random annotated programs that have an already known verification outcome. To achieve this, XDsmith focuses on a subset of Dafny and specific annotations, e.g., of the form $x == v$ such that they are satisfied or violated for a specific program execution.

We are not aware of program-generation techniques that target analyzers reasoning about reachability properties, like Minotaur. In general, a downside of these techniques is that the generated programs must be restricted. Minotaur, in particular, generates programs with a specific structure and supports a subset of C (although the latter is tailored to the analyzers under test, as discussed).

Specification-based testing. Specification-based testing techniques require a specification of correct behavior for an analyzer under test; they then test the analyzer, for example using fuzzing, against this specification. Such techniques have, for instance, been used to test implementations of dataflow analyses [37], abstract domains [7], or certain lattice properties of abstract interpreters [32].

In contrast to Minotaur, these techniques require time and manual effort to write the specifications, which can often be nontrivial for analyzers. For instance, the work on testing dataflow analyses [37] uses an independent SMT-based dataflow analysis, developed by the authors, as a specification. Due to the comparison with the SMT-based dataflow analysis, this approach could also be seen as an instance of differential testing. In fact, differential testing, which we discuss next, could be viewed as specification-based testing, where the specification is another implementation.

Differential testing. Differential testing of analyzers consists of running multiple of them on the same input programs and comparing their results. If they disagree, then an issue is detected in at least one of them; however, both the type of issue (soundness or precision) and which analyzer exhibits the issue remain unclear. Such techniques have been used to, e.g., test C compilers [40] and analyzers [12, 15, 26], SMT solvers [34, 38], and a symbolic-execution engine [25].

In contrast, Minotaur does not require more than one analyzer under test and is able to classify any detected issue as soundness or precision related. In addition, differential testing can suffer from false positives when comparing analyzers that implement intentionally different analysis techniques, e.g., when comparing analyzers that are intentionally imprecise, such as abstract interpreters, with analyzers that are intentionally unsound, such as bounded model checkers.

Metamorphic testing. Metamorphic-testing techniques typically run an analyzer on an input program to get its result; based on this, they then transform the program such that the expected analyzer result for the transformed program is known. An issue is detected when the actual analyzer result does not match the expected one. Such techniques have been applied to test SMT solvers [30, 39], Datalog engines [29, 31], static analyzers [19, 33, 41, 42], etc.

Metamorphic testing detects an issue on a pair of programs (original and transformed), making it unclear whether the issue is soundness or precision related. This is not the case with Minotaur. He et al. [19], in addition to a static, metamorphic oracle, define a dynamic one that is able to distinguish soundness and precision issues by comparing with concrete program executions.

8 Conclusion

In this paper, we presented a novel testing framework to find soundness and precision issues in a wide range of program analyzers. Our approach generates unsafe (respectively safe) programs from satisfiable (respectively unsatisfiable) SMT constraints. Minotaur reports an issue if an analyzer returns a result that does not match the ground truth. By knowing the ground truth, as given by the satisfiability of the used constraint, Minotaur can also indicate whether the issue is related to precision or soundness. Overall, Minotaur found 19 soundness and 11 precision issues in 11 analyzers.

Acknowledgments

We thank Samuel Pilz and the anonymous reviewers for their insightful and constructive feedback. This work was supported by Maria Christakis' ERC Starting grant 101076510.

References

[1] [n. d.]. The International Satisfiability Modulo Theories Competition. https://smt-comp.github.io.
[2] Lukas Atkinson and Michael Förderer. [n. d.]. gcovr. https://gcovr.com/en/stable.
[3] Dirk Beyer. 2012. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org.
[4] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV (LNCS, Vol. 6806)*. Springer, 184–190.
[5] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-Invariants and k-Induction. In *SAS (LNCS, Vol. 9291)*. Springer, 145–161.
[6] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.

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

[8] 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.

[9] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *ICSE*. ACM, 144–155.

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

[11] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. 2009. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *ASE*. IEEE Computer Society, 137–148.

[12] 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.

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

[14] Evren Ermis, Alexander Nutz, Daniel Dietsch, Jochen Hoenicke, and Andreas Podelski. 2014. Ultimate Kojak - (Competition Contribution). In *TACAS (LNCS, Vol. 8413)*. Springer, 421–423.

[15] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ISSTA*. ACM, 1219–1231.

[16] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound Sequentialization for Concurrent Program Verification. In *PLDI*. ACM, 506–521.

[17] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. 2017. Loop Invariants from Counterexamples. In *SAS (LNCS, Vol. 10422)*. Springer, 128–147.

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

[19] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *PACMSE* 1 (2024), 1656–1678. Issue FSE.

[20] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *SAS (LNCS, Vol. 5673)*. Springer, 69–85.

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

[22] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *ISSTA*. ACM, 556–567.

[23] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV (LNCS, Vol. 5643)*. Springer, 661–667.

[24] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *VSTTE (LNCS, Vol. 12031)*. Springer, 1–18.

[25] 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.

[26] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.

[27] Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *ASE*. ACM, 45:1–45:12.

[28] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370.

[29] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *ESEC/FSE*. ACM, 639–650.

[30] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.

[31] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.

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

[33] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *ICSE*. IEEE Computer Society, 550–562.

[34] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *PACMPL* 5 (2021), 1–19. Issue OOPSLA.

[35] Jiri Slaby, Jan Strejcek, and Marek Trtík. 2012. Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution. In *FMICS (LNCS, Vol. 7437)*. Springer, 207–221.

[36] Jiri Slaby, Jan Strejcek, and Marek Trtík. 2013. Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution—(Competition Contribution). In *TACAS (LNCS, Vol. 7795)*. Springer, 630–632.

[37] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.

[38] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.

[39] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.

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

[41] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.

[42] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *ESEC/FSE*. ACM, 237–249.