Interrogation Testing of Program Analyzers for Soundness and Precision Issues

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 are critical in safeguarding software reliability. However, due to their inherent complexity, they are likely to contain bugs themselves, and the question of how to detect them arises. Existing approaches, primarily based on specification-based, differential, or metamorphic testing, have been successful in finding analyzer bugs, but also come with certain limitations.

In this paper, we present *interrogation testing*, a novel testing methodology for program analyzers, to address limitations in existing metamorphic-testing techniques. Specifically, interrogation testing introduces two key innovations by (1) incorporating more information from analyzer queries to construct more powerful oracles, and (2) introducing a knowledge base that maintains a history of diverse queries. We implemented interrogation testing in Sherlock and tested 8 mature analyzers—including model checkers, abstract interpreters, and symbolic-execution engines—that can prove the safety of assertions in C/C++ programs. We found 24 unique issues in these analyzers, 16 of which are soundness related, i.e., an analyzer does not report an assertion that can be violated. Our experimental evaluation demonstrates Sherlock's effectiveness by finding issues between 7x and 906x faster than our baseline, which is inspired by the state of the art.

CCS Concepts

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

Keywords

interrogation testing, program analyzers, unsoundness, imprecision

ACM Reference Format:
David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. 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/10.1145/3691620.3695034

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

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695034

1 Introduction

There is an abundance of program analyzers that are developed both in academia and industry with the purpose of guarding the reliability of modern-day software. But how can we guard the guards themselves? Program analyzers are highly complex tools, implementing sophisticated algorithms and performance optimizations. In addition, analyzers typically integrate several self-contained, core analysis components, such as specialized solvers, which are complex as well. Due to this overall complexity, program analyzers are all the more likely to contain reliability issues themselves.

The most important reliability issues in analyzers are soundness and precision issues. The former are the most critical since they may cause an analyzer to return 'correct' for incorrect software (false negative). False negatives can have disastrous consequences when analyzing safety-critical software. The latter are also important; they may cause an analyzer to return 'incorrect' for correct software (false positive). False positives are cumbersome and time-consuming for users to distinguish, and perhaps more importantly, they may discourage users from taking true positives seriously or from using program analysis altogether [14].

State of the art and challenges. Although it is possible to prove properties about the design of a program analysis, verifying the absence of issues in its implementation is extremely demanding (see, for instance, [27, 29] for long-term efforts to fully verify large systems). In contrast to verification, automated test-generation techniques have been used to detect issues in program analyzers without providing absolute correctness guarantees. However, each of the existing testing techniques comes with its own limitations.

Program generation, in the context of testing analyzers, aims to produce programs such that their correctness is known by construction (e.g., [8, 23]). The analyzers are then run on these programs, and if they contradict the expectation about a program's correctness, an issue is detected. In the general case however, knowing what to anticipate from a program analyzer on a given input is undecidable. For this reason, such approaches typically restrict the diversity and complexity of the generated programs.

Specification-based testing (e.g., [9, 33, 40]) involves specifying the correct behavior of the analyzer and testing it, for instance using fuzzing, against this provided specification. Similar to verification,

fully specifying large systems is highly non-trivial and requires prohibitive amounts of time and effort.

Differential testing of program analyzers (e.g., [18, 25, 28, 35, 41]) constitutes running multiple of them on the same input program and comparing their responses for disagreement. If a disagreement exists, then at least one analyzer must be wrong. A benefit of this approach is that it does not require a specification of correct behavior. However, differential testing is not suitable for emerging domains where there do not yet exist many (or even two) analyzers, or for domains without standardization where no two analyzers accept the same input. For instance, this is the case with Datalog engines, where many tools support different Datalog dialects. Moreover, when using differential testing, it is unclear which analyzer is to blame for a disagreement, e.g., it could be that the majority is wrong and a single analyzer correct.

Metamorphic testing (e.g., [20, 30–32, 34, 42, 44–46]) transforms an input program such that the expected analysis response is known. For example, consider an analyzer and a program that it finds incorrect. A metamorphic transformation could add dead code to the program. Then, the resulting program should still be found incorrect by the analyzer. If this is not the case, an issue has been found. In general, metamorphic testing lies in the middle of the spectrum between specification-based and differential testing. It only requires defining metamorphic transformations, which is much easier than writing full-blown specifications, and it does not require the existence of multiple analyzers accepting the same input.

However, the metamorphic transformations that can be defined are restricted, thereby limiting the effectiveness of this approach. In particular, metamorphic testing of program analyzers typically consists of the following workflow: query the analyzer on an input and get its response (i.e., the input is correct or incorrect); based on the response, transform the input such that the new expected response is known; query the analyzer again on the transformed input and get the new actual response; detect a bug if the new actual response does not match the expected one.

Our approach. In this paper, we propose *interrogation testing* of program analyzers, which is at least as powerful as metamorphic testing. Consider a suspect-interrogation analogy, where the goal is to force the suspect into a contradiction. Metamorphic testing resembles having a new investigator interrogate the suspect after every two yes/no questions. Interrogation testing, on the other hand, is like having the same investigator carry out the entire interrogation and ask for more details than yes/no responses.

Specifically, there are two key innovations. First, interrogation testing may use more information than the analyzer response in order to form queries. Second, it integrates a knowledge base that enables maintaining a (long) history of past analyzer queries (including the analyzer's response) to form more diverse queries in the future. With these innovations, we can design more sophisticated and challenging queries for the program analyzers under test. In turn, this allows detecting more soundness and precision issues.

We instantiate interrogation testing in a technique for testing program analyzers that reason about reachability, such as abstract interpreters, bounded model checkers, and symbolic-execution engines. We implement our technique in a tool called Sherlock, which we used to test 8 analyzers, namely, CBMC [15], Clam [19], CPAchecker [6], ESBMC [16], KLEE [10], MOPSA [24], Symbiotic [37, 38], and UAutomizer [21, 22].

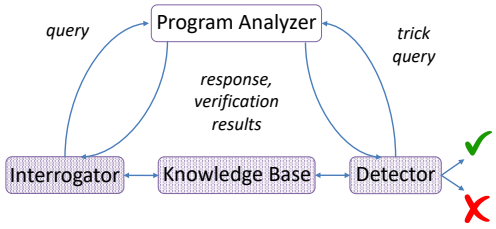

Figure 1: Overview of interrogation testing.

Contributions. Our paper makes the following contributions:

(1) We introduce interrogation testing, a general methodology for testing program analyzers, which subsumes existing metamorphic-testing techniques.
(2) We instantiate interrogation testing in a technique for testing analyzers that reason about reachability.
(3) We implement our technique in a tool called Sherlock.
(4) We evaluate Sherlock by testing 8 mature analyzers. Sherlock found 24 unique issues, 23 of which were confirmed by the analyzer developers. Of the analyzers we tested, 7 contained a total of 16 soundness issues.

Data availability. We provide our tool and documentation at: https://github.com/Rigorous-Software-Engineering/sherlock

Outline. In Sect. 2, we present interrogation testing, and in Sect. 3, we instantiate it in a technique for testing program analyzers that reason about reachability. Sect. 4 describes implementation aspects of Sherlock. We present our experimental evaluation in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

2 Interrogation Testing Overview

On a high level, interrogation testing aims to force the analyzer under test into a contradiction. Finding a contradiction means that a soundness or precision issue is detected. To this end, our methodology may use not only the analyzer's Boolean response (i.e., whether the analyzer found the analyzed program to be correct), but also its more fine-grained verification results (e.g., the concrete set of failing assertions) to both expose contradictions and form new queries. It also maintains a history of responses and verification results from previous queries, enabling the construction of more diverse queries over time. Fig. 1 shows an overview of interrogation testing.

Given a program analyzer under interrogation, an *interrogator* starts by posing a query to the analyzer requesting its response on the correctness of an input. As an example, consider an analyzer that reasons about reachability of error locations. If no error location is reachable, the program is found correct, otherwise it is found incorrect. In addition to the analyzer response, the interrogator may request the corresponding verification results. In the above example, the verification results are the specific error locations that are (un)reachable. Generally, *verification results* refer to the verification status of individual specifications (e.g., pre- and post-conditions, assertions, or invariants) in the program.

The interrogator remembers the analyzer response and its verification results by storing them in a *knowledge base*. Such an interrogation round may be repeated any number of times, and the

interrogator may use the knowledge base for constructing further, more diverse queries. For instance, the interrogator could randomly mutate queries from the knowledge base to derive new queries, and thereby, enrich the knowledge base.

At any point during the interrogation, a *detector* may pose a *trick query* to the analyzer. A trick query differs from other queries in that the expected response and potentially also the corresponding verification results are known a-priori, i.e., they can be derived from the knowledge base. If the analyzer contradicts what is expected, then an issue has been found. For example, a program that was found incorrect by the above analyzer should remain incorrect when extending it with, say, additional, unreachable error locations. Moreover, the reachable error locations of the extended program should exactly match those of the original program.

Although these example trick queries might appear too simple to be able to uncover issues in analyzers, consider that existing work (e.g., [30, 31, 42, 44]) has already detected a few such issues with much more restricted approaches, i.e., without verification results or access to previous queries through the knowledge base.

Algorithm. Alg. 1 describes the general interrogation-testing algorithm. It takes as input an analyzer a and a set of seed input programs *seeds* (line 1). Each *seed* can be an existing program, e.g., from an open-source project, or automatically generated, e.g., using CSmith [43] or similar tools. For each *seed* (line 2), we initialize an empty knowledge base kb (line 3), which remembers queries to the analyzer under test, the analyzer's responses, and its verification results. The interrogator first queries the analyzer with *seed* to get its response r_{seed} and verification results vr_{seed} (line 4), which it then stores in the knowledge base (line 5).

As long as there is budget (line 6), which is specified by the user, we interrogate the analyzer as follows. We select a query x with the corresponding analyzer response r_x and verification results vr_x from the knowledge base (line 7). The selection may be performed uniformly at random or using any other distribution that prioritizes more interesting queries. Using x, r_x, vr_x, we generate a new query y together with the expected analyzer response $\overline{r_y}$ and verification results $\overline{vr_y}$ (line 8). If y is not a trick query, that is, if we do not need to know the expected analyzer response and verification results, $\overline{r_y}$ and $\overline{vr_y}$ are empty. On line 9, we pose query y to the analyzer and get its actual response r_y and verification results vr_y. Next, if y, r_y, vr_y achieves more diversity, we store it in the knowledge base.

Observe that we use a diversity criterion for adding a new entry in the knowledge base. This criterion could be trivially true, thereby allowing all generated queries to be stored. The downside of such a permissive criterion is that the knowledge base would constantly grow, resulting in clusters of similar entries. As a result, when selecting a query from the knowledge base, it becomes more likely to pick from one of the large clusters and grow them even more. Alternative diversity criteria include checking whether new code coverage of the analyzer under test or new verification results are achieved. We discuss our instantiation of the diversity criterion in the next section.

If y is a trick query, we check the oracle (line 12). Specifically, we check whether the actual response and verification results match the expected ones. If they do not, then the algorithm reports that an issue has been found in analyzer a. Note that the algorithm does not determine the type of issue, i.e., soundness or precision. For instance, consider the earlier example of a program, say x, which was found incorrect by an analyzer, and an extended version of x, say y, with additional unreachable error locations. If the reachable error locations of y do not exactly match those of x because, say, more error locations are found to be reachable for y, the oracle fails. However, our algorithm cannot know if analyzer a missed a reachable error location in x (soundness issue) or if it reports an unreachable error location in y as reachable (precision issue).

Interrogation testing may be used to test analyzers other than those reasoning about reachability, despite our focus on reachability in this paper. Consider Datalog engines, which compute data-flow information. For each response, Datalog engines typically record provenance, which, similar to verification results, shows how the response is derived. Of course, knowing how data flows to the program output allows for generating more sophisticated queries than when only knowing the output. Another example are SMT solvers, which compute proof artifacts, such as a model for satisfiable formulas or an unsatisfiable core for unsatisfiable formulas. Again, using

Algorithm 1: Interrogation testing.

```
1  function INTERROGATE(a, seeds):
       /* for each seed input in seeds                                   */
2      foreach seed ∈ seeds do
           /* initialize an empty knowledge base kb                     */
3          kb := INITKNOWLEDGEBASE();
           /* pose query seed to analyzer a                             */
           /* get response r_seed and verification results vr_seed      */
4          r_seed, vr_seed := POSEQUERY(seed, a);
           /* store query in knowledge base                             */
5          kb.REMEMBERQUERY(seed, r_seed, vr_seed);
6          while ¬BUDGETDEPLETED() do
               /* get random query from knowledge base                  */
               /* query distribution may be non-uniform                 */
7              x, r_x, vr_x := kb.GETRANDQUERY();
               /* generate random query from existing query             */
               /* predict expected response r_y and results vr_y        */
8              y, r_y, vr_y := kb.GENRANDQUERY(x, r_x, vr_x);
               /* pose new query y                                      */
               /* get actual response r_y and results vr_y              */
9              r_y, vr_y := POSEQUERY(y, a);
               /* if new query achieves more diversity                  */
10             if kb.ACHIEVESMOREDIVERSITY(y, r_y, vr_y) then
                   /* store query in knowledge base                     */
11                 kb.REMEMBERQUERY(y, r_y, vr_y);
               /* if new query is trick query, i.e.,                    */
               /* expected response and results are not empty,          */
               /* and if oracle does not hold                           */
12             if ISTRICKQUERY(y, r_y, vr_y) ∧
                  ORACLEFAILS(r_y, r_y, vr_y, vr_y) then
13                 REPORTISSUE(x, y, a);
```

```c
int main() {
  float f = ★;
  assert(f < 1.0 || f > -1.0);
  assume(f > 0.0);
  printf("%f\n", f);
  assert(f > 0.0);
  // assert(true);
  return 0;
}
```

(a) Soundness issue in ESBMC.

```c
void main() {
  int z = ★;
  int k = ★;
  assume(1 < z);
  assume(1 <= z && k <= 1);
  // assume(1 < z && k <= 1);
  assert(0);
}
```

(b) Soundness issue in CBMC.

```c
int main() {
  int s = ★;
  assume(s != 0 && s <= 1);
  assume(s > 1);
  assert(0);
  assume(s < 2);
  return 0;
}
```

(c) Precision issue in Clam.

Figure 2: Example soundness and precision issues detected by Sherlock and fixed by the analyzer developers.

these artifacts makes it possible to create more complex queries than when only knowing a formula's satisfiability.

3 Interrogation Testing in Sherlock

As mentioned earlier, we instantiate interrogation testing in a technique for testing program analyzers that derive reachability of error locations. Examples of such analyzers include abstract interpreters, bounded model checkers, and symbolic-execution engines. We implemented our technique in a tool called Sherlock, which we used to test 8 analyzers and found 24 issues in these analyzers.

The analyzers we target accept C/C++ programs and understand assertions and assumptions (expressed using `assume` statements or similar constructs). At runtime, executions that violate an asserted or assumed condition are immediately terminated. However, only assertion violations are considered "errors" and are, thus, reported by analyzers. In this context, given a program annotated with assertions and assumptions, the analyzer response refers to whether the program is found correct or incorrect, and the verification results consist in which assertions are proved correct and which are not.

On a high level, Sherlock produces trick queries by adding or mutating assertions and assumptions in a program x to generate a transformed program y. The transformations are designed such that the expected analyzer response and verification results for y are easily derived from those of x. For example, when generating y by strengthening an assumed condition in x, fewer program executions are allowed through the assumption, fewer error locations might be reachable, and therefore, fewer assertions might fail. So, the failing assertions in x should contain the failing assertions in y. Similarly, when generating y by converting a verified assertion in x into an assumption, no new error locations are made reachable, and thus, the sets of failing assertions for x and y should be the same.

In the rest of this section, we provide examples of soundness and precision issues that Sherlock detected using interrogation testing. We also describe in detail the program transformations that Sherlock performs based on the analyzer response and verification results as well as the diversity criterion we use for adding new entries in the knowledge base.

3.1 Examples of Detected Issues

Fig. 2 shows three soundness and precision issues that Sherlock detected in ESBMC, CBMC, and Clam. All three issues were confirmed and fixed by the analyzer developers. Note that we reported all issues anonymously; in the figure, each sub-caption links to the corresponding bug report.

ESBMC. Fig. 2a shows a program that revealed a soundness issue in ESBMC. Variable `f` is a non-deterministic float, denoted by ★ (line 2). The assertion on line 3 may fail as `f` may have value NaN (Not a Number). The assertion on line 6 never fails due to the assumption on line 4. ESBMC misses the potential assertion failure in this program and unsoundly proves it correct. Interestingly, when dropping line 5, ESBMC does report the assertion violation.

Sherlock detects this issue by comparing the analyzer response and verification results for the program in Fig. 2a (let us refer to it as program x) with those for a program where the assertion on line 6 is replaced with the weaker assertion on line 7 (program y). The ESBMC response for x is correct, and the verification results are that both assertions (lines 3 and 6) are verified. When an assertion is verified, all executions through the assertion must satisfy its condition. Consequently, all executions through the assertion should also satisfy a weaker condition.

This is exactly the trick query that Sherlock generates from program x: since the assertion on line 6 is verified, its condition could be replaced with a weaker one in program y, and the failing assertions in x and y should be exactly the same. In this case, Sherlock randomly generates the weaker condition `true`, but any other weaker condition could also be generated, e.g., `f > -2.0`. The analyzer response for y is incorrect, and the verification results are that the assertion on line 3 fails. As a result, Sherlock detects an issue, which, after manual inspection and confirmation from the analyzer developers, we classify as a soundness issue.

Note that x and y are minimized versions of the programs that actually revealed this soundness issue (see Sect. 4), which are in turn generated by performing a series of (trick) queries on an existing or automatically generated seed program. The seed program may be free of any assertions or assumptions.

CBMC. Fig. 2b shows a program that revealed a soundness issue in CBMC. Variables `z` and `k` are assigned non-deterministic integer values (lines 2–3). On lines 4–5, we assume properties about the values of these variables, and on line 7, we have an assertion that always fails when reachable. Program x is the program in Fig. 2b when replacing the assumption on line 5 with the one on line 6. For x, CBMC reports the assertion failure.

Sherlock generates program y by weakening the assumption on line 6 to `assume(1 <= z && k <= 1)`. When weakening an assumption, more program executions are allowed through it, more error locations might be reachable, and thus, more assertions might

fail. So, the failing assertions in y should contain the failing assertions in x. However, CBMC does not report any assertion violation for y, and Sherlock detects the issue. The analyzer developers explained that, instead of assuming `1 <= z && k <= 1` on line 5, CBMC incorrectly assumes `z == 1`, which in conjunction with the assumption on line 4 makes the assertion unreachable.

Clam. Fig. 2c shows a program y that revealed a precision issue in Clam. The assumptions on lines 3–4 make the assertion on line 5 unreachable. Despite this, Clam imprecisely reports an assertion violation. To generate this program y, Sherlock added the assumption on line 6. For x (without line 6), Clam is precise and does not report an assertion violation. Conversely to the trick query revealing the soundness issue in CBMC, when adding or strengthening an assumption in x to generate y, the failing assertions in x should contain the failing assertions in y. In particular, when adding or strengthening an assumption, fewer program executions are allowed through it, fewer error locations might be reachable, and thus, fewer assertions might fail. This is not the case here, and Sherlock detects the issue.

3.2 Interrogator

In Sherlock, the interrogator is used to bootstrap the knowledge base for each seed program (see lines 4–5 in Alg. 1). In the future, we plan to experiment with alternative interrogation strategies.

3.3 Detector

We now describe in detail how the detector constructs trick queries.

Trick queries. In interrogation testing, trick queries serve a dual purpose: (a) they contribute in expanding the knowledge base with new entries, and (b) potentially reveal soundness and precision issues in the analyzer under test. As mentioned earlier, given a program x, Sherlock generates a program y by adding or mutating assertions and assumptions in x. The transformations are designed so that we can derive the expected analyzer response and verification results for y by knowing those for x. Then, the oracle for correct (with respect to soundness and precision) analyzer behavior is that the actual analyzer response and verification results for y match the expected ones.

We now describe our transformations and their respective oracles in detail (see the last two columns of Tab. 1 for an overview). Note that, in our oracles, the set of failing assertions in x is denoted by F_x and the set of failing assertions in y by F_y. Analyzers do not typically report failing *assumptions*; we, therefore, do not use failing assumptions in our oracles.

AddAssert: Given a program x and a location l in x, program y is generated by adding a random assertion s, which may or may not fail, at l. (Although s is generated randomly, Sherlock takes into account live variables at l, their types, and constants in the program.) The oracle for this transformation is $F_y \setminus \{s\} \subseteq F_x$. It covers the following cases: (a) if s does not fail, we expect $F_y = F_x$, and (b) if s fails, other, previously failing assertions may now hold due to the additional constraints from s; consequently, except for s, there should be no additional failing assertions in program y.

AddAssume: Given a program x and a location l in x, program y is generated by adding a random assumption, which may or may not hold, at l. The oracle for this transformation is $F_y \subseteq F_x$. Similar to the previous transformation, it covers the following cases: (a) if the assumption holds, we expect $F_y = F_x$, and (b) if not, other, previously failing assertions may now hold due to the additional constraints from the assumption.

Table 1: The transformations applied by Sherlock and their respective oracles.

Type	Transformation	Oracle
Strengthen	AddAssert	$F_y \setminus \{s\} \subseteq F_x$
	AddAssume	$F_y \subseteq F_x$
	StrengthenAssert	$F_y \setminus \{s\} \subseteq F_x$
	StrengthenAssume	$F_y \subseteq F_x$
Weaken	WeakenAssume	$F_x \subseteq F_y$
	WeakenAssert	$F_x \setminus \{s\} \subseteq F_y$
Even	WeakenInvariant	$F_x = F_y$
	AssumeInvariant	$F_x = F_y$

StrengthenAssert: Given a program x and a location l in x with an assertion s, program y is generated by potentially strengthening the asserted condition P of s. For instance, P may be transformed into $P \wedge P'$, where P' is randomly generated (again taking into account live variables at l, their types, and constants in the program). Since P' can be *true*, the transformed condition may be equivalent to P. The oracle for this transformation is $F_y \setminus \{s\} \subseteq F_x$. It covers the following cases: (a) if s fails in x, s must also fail in y, and we expect $F_y = F_x$, (b) if s fails neither in x nor in y, we again expect $F_y = F_x$, and (c) if s does not fail in x but does fail in y, other, previously failing assertions may now hold in y due to the stronger constraints; consequently, except for s, there should be no additional failing assertions in y.

StrengthenAssume: Given a program x and a location l in x with an assumption, program y is generated by potentially strengthening the assumed condition. Similar to AddAssume, the oracle for this transformation is $F_y \subseteq F_x$.

WeakenAssert: In contrast to StrengthenAssert, this transformation potentially weakens the asserted condition. The oracle for this transformation is $F_x \setminus \{s\} \subseteq F_y$. It covers the following cases: (a) if s does not fail in x, s must also not fail in y, and we expect $F_y = F_x$, (b) if s fails in x, other, previously verified assertions may now fail in y due to the weaker constraints.

WeakenAssume: In contrast to StrengthenAssume, this transformation potentially weakens the assumed condition. Hence, the oracle for this transformation is $F_x \subseteq F_y$.

WeakenInvariant: In contrast to WeakenAssert, this transformation is only applied if the assertion s has been verified (by the analyzer under test) in x, i.e., it is an invariant. The oracle for this transformation is $F_x = F_y$. Since the analyzer claims that s does not fail in x, the oracle is equivalent to case (a) from WeakenAssert.

AssumeInvariant: Given a program x and a location l in x with an assertion s that has been verified (by the analyzer under test), program y is generated by converting s into an assumption. The oracle for this transformation is $F_x = F_y$. Since the analyzer claims that s does not fail in x, converting s into an assumption should not affect the set of failing assertions.

Recall that, in Fig. 2a, both assertions (lines 3 and 6) are verified by ESBMC. We can then apply WeakenInvariant on line 6 to weaken the asserted condition to `true` (line 7). For the generated program however, ESBMC warns that the assertion on line 3 fails, which violates the oracle for this transformation, i.e., the set of failing assertions for the two programs should be the same. Similarly, in Fig. 2b, we can apply WeakenAssume on line 6 to obtain line 5, and in Fig. 2c, we can apply AddAssume to obtain line 6.

Note that we generate random assertions and assumptions by producing random Boolean conditions; for instance, (in-)equalities, with random expressions as operands. The expressions are over initialized numerical variables that are in scope, constants from the program, and random constants.

Stacked transformations. To increase the chances of detecting issues in the analyzer under test, Sherlock stacks the above transformations, that is, it applies multiple transformations to program x when generating y. Note, however, that transformations are only stacked when they share the same generic oracle type.

As shown in the first column of Tab. 1, we categorize our transformations into three oracle-based types, Strengthen, Weaken, and Even. The oracle for transformations of type Strengthen is $F_y \setminus M_{x,y} \subseteq F_x$, where $M_{x,y}$ denotes the set of all assertions that are added or mutated when transforming x to y; the oracle for Weaken is $F_x \setminus M_{x,y} \subseteq F_y$; and the oracle for Even is $F_x = F_y$.

For example, to generate the program in Fig. 2a, we can first apply stacked transformations of type Strengthen, which add assertions and assumptions as well as strengthen them. The generated program together with the analyzer response and its verification results can then be stored in the knowledge base. When it is selected for transformation, we can apply Even, which weakens the invariant on line 6 and reveals the soundness issue.

Transformation strategy. Alg. 2 describes in more detail how Sherlock applies stacked transformations. In particular, it shows the implementation of function genRandQuery, which is called on line 8 of Alg. 1. The function takes as input a program x as well as the analyzer response r_x and verification results vr_x for x (line 1). It returns a transformed program y together with the expected analyzer response $\overline{r_y}$ and verification results $\overline{vr_y}$.

On line 2, we choose the number of transformations to apply, *numTr*, according to a maximum-exponent setting, *maxExp*. In our experiments, we set *maxExp* to 5 and, therefore, may apply up to 32 transformations. This is inspired by how AFL [3] chooses the number of stacked mutations to apply to an input. On line 3, we define the maximum number of failed transformation attempts, *numFailAtt*, that we tolerate based on an attempt-factor setting, *attFctr*. In our experiments, we set *attFctr* to 10. (We explain in which cases failed attempts occur and provide examples in the following.) Next, we choose a random transformation type (line 4) and initialize the return variables (line 5).

While the number of failed attempts has not been depleted and there are more transformations to apply (line 6), we choose a random location, l, in y to transform (line 7). On line 8, we call doRandTr to apply a transformation of type *trType* to location l in y. In Sherlock, the knowledge base *kb* is, for instance, used to check if the WeakenInvariant transformation can be applied at the selected location. Note that *trType* determines the oracle, and thus, the expected analyzer response and verification results for y.

Algorithm 2: Transformation strategy in Sherlock.

```
1  function genRandQuery(x, r_x, vr_x):
       /* choose number of transformations to apply */
2      numTr := 2^randInt(0, maxExp);
       /* compute number of failed attempts to transform */
3      numFailAtt := attFctr * numTr;
       /* choose transformation type */
4      trType := rand({Strengthen, Weaken, Even});
       /* initialize return variables */
5      y, r_y, vr_y := x, r_x, vr_x;
6      while numFailAtt > 0 ∧ numTr > 0 do
           /* choose location to transform */
7          l := y.locs[randInt(0, len(y.locs))];
           /* try to apply random transformation */
8          y', r_y', vr_y', tr := kb.doRandTr(y, r_y, vr_y, l, trType);
           /* if failed to apply */
9          if tr = NULL then
               /* update the number of failed attempts */
10             numFailAtt := numFailAtt − 1;
11             continue;
           /* if applied successfully */
           /* update the number of transformations */
12         numTr := numTr − 1;
           /* update return variables */
13         y, r_y, vr_y := y', r_y', vr_y';
14     return y, r_y, vr_y;
```

However, we may fail to apply a certain transformation to a particular location l (line 9). For example, transformations of type Even require that l contains a verified assertion. In case of a failed transformation attempt, we decrement *numFailAtt* (line 10) and continue with the next loop iteration (line 11). If the transformation is applied successfully, we decrement the number of transformations (line 12) and update the return variables (line 13).

3.4 Knowledge Base

On a high level, the knowledge base maintains a history of previous, diverse queries together with the corresponding analyzer response and verification results. It is partially inspired by how greybox fuzzers maintain a corpus of interesting inputs.

Diversity criterion. To control the size and diversity of the knowledge base, we introduce a diversity criterion, which serves as a knowledge-base gatekeeper. In particular, we use this criterion to determine if an analyzer query should be added to the knowledge base. This is similar to how greybox fuzzers only add inputs to the corpus if they increase coverage. The goal is to prevent the formation of large clusters of similar entries, which bias the search.

Specifically, after each analyzer query, we track, for each location in program p, whether it contains an assertion that the analyzer does not verify. Note that we track the locations in p with respect to the original *seed* (line 2 of Alg. 1), that is, our transformations do not affect program locations. In other words, for each p, we obtain a

diversity profile, which is encoded as a mapping $seed.locs \mapsto \{0, 1\}$. If a program p has the same diversity profile as any program in the knowledge base, then we do not add p to the knowledge base.

Time to live. For our diversity criterion, the maximum number of entries in the knowledge base is $2^{seed.locs}$. Consequently, for small seed programs, we might quickly fill the knowledge base. From this point on, diverse and potentially rare queries, which could lead to detecting analyzer issues, are no longer added to the knowledge base. As a result, some interesting areas of the search space are not explored effectively (thereby reducing query diversity) since the knowledge base has reached a "local optimum".

To address this, we assign a time-to-live (TTL) attribute to each program in the knowledge base and replace the program once its TTL has expired. More specifically, whenever a program a is selected for further transformation, we decrement its TTL. Once we have a program b that has the same diversity profile as a and a's TTL has expired, we replace a with b in the knowledge base. As our experiments show, this mechanism increases the effectiveness of Sherlock in detecting analyzer issues. It is partially inspired by optimization techniques that try to escape local optima, for instance, by restarting the search from a different point in the search space.

3.5 Discussion

For Sherlock, we assume that the analyzers under test reason about reachability of assertions and understand assumptions. Note that assertions may also be expressed as if-statements guarding an error (e.g., overflow, null-pointer dereference, etc.) and assumptions as if-statements that successfully terminate the program. Consequently, it is actually not even necessary for analyzers to support explicit `assert` and `assume` statements.

In addition, Sherlock assumes that the analyzers report all reachable assertion violations in an analyzed program. Note, however, that Sherlock can also handle analyzers that stop after reporting the first assertion violation by ensuring there is a single assertion in the program. For programs with multiple assertions, one can perform multiple analyzer queries, each with a single assertion.

As for interrogation testing in general, the above assumptions can be relaxed further depending on the type of tested analyzers.

4 Implementation

In order to increase the chances that the issues we report are fixed, we developed the following two improvements in Sherlock.

Issue prioritization. When testing program analyzers that may be imprecise, such as abstract interpreters, we found that Sherlock may report several, potential precision issues. In our experience, however, analyzer developers do not value these as much as soundness issues and are less likely to fix them. After all, these analyzers are usually designed to trade precision for other qualities, such as scalability. When reporting issues for analyzers with known imprecision, we therefore prefer to prioritize soundness issues. For this, we use differential testing with analyzers that are not designed to be imprecise, such as model checkers. More specifically, when an imprecise analyzer exhibits an issue for a pair of programs, we also run a (more) precise analyzer on the corresponding programs. If the precise analyzer reports more warnings for any of the two programs, we prioritize the issue; it is likely a soundness issue.

Issue minimization. Once an issue is detected, we try to minimize the program(s) that exhibit it before reporting it to the developers. This step is important for developers as it may significantly facilitate debugging, and thus, improve their response time. For issue minimization, we use C-Reduce [1], a popular C/C++ program reducer. C-Reduce takes as input a program that has a particular property of interest (e.g., it triggers a compiler crash) and iteratively produces a much smaller program that exhibits the same property.

We adapt C-Reduce as follows. Consider an issue that is detected by Sherlock when applying stacked transformations T on program x to obtain program y; assume that x and y (i.e., their set of failing assertions) violate oracle O. We use C-Reduce for reducing x to x' such that, when we re-apply transformations T to x', we obtain y', and x' and y' violate oracle O. Note that we do not allow the reduction of assertions or assumptions that are affected by T. For instance, if the condition of an assumption is reduced, re-applying our transformation might yield a completely different condition or not be possible at all. Nevertheless, C-Reduce may reduce all other assertions and assumptions. This minimization technique has been very effective in practice; we have observed reductions of up to 300x in lines of code (e.g., for issue 13 in Tab. 2).

5 Experimental Evaluation

In this section, we address the following research questions:

RQ1: How effective is Sherlock in detecting analyzer issues?
RQ2: What are characteristics of the detected issues?
RQ3: Which transformations are effective in detecting issues?
RQ4: How does Sherlock compare to metamorphic testing?
RQ5: Is the knowledge base effective in detecting issues?

5.1 Setup

We used Sherlock to test 8 well known and publicly available analyzers, namely, CBMC, Clam, CPAchecker, ESBMC, KLEE, MOPSA, Symbiotic, and UAutomizer, that use a wide range of different program-analysis techniques, including abstract interpretation, model checking, and symbolic execution. We use the following two setups for our experiments with these analyzers.

Issue detection: To detect analyzer issues (RQ1–2), we collected seeds (see inputs of Alg. 1) from several sources, namely, the Competition on Software Verification (SV-COMP) [4], the Competition on Software Testing (Test-COMP) [5], and the open-source repository of the Goblint static analyzer [2]. We also generated seeds using Csmith [43]. In total, we assembled a set of 4500 seeds. For each seed, the budget is depleted (see Alg. 1) (1) if no analyzer query is added to the knowledge base after 100 consecutive queries, or (2) after 2000 queries. We use a time limit of 30 secs for each analyzer query.

Issue reproduction: With this setup, we aim to re-find already detected issues (RQ3–5) and evaluate the performance of Sherlock. Here, the *seeds* of Alg. 1 contain the single seed that originally revealed the issue. The budget of Alg. 1 is seven days unless stated otherwise. Note, however, that we cannot be sure that each detected issue corresponds to the original issue we want to re-find. After all, an analyzer may have several issues. For this reason, we only focus on fixed issues: for each detected issue, we apply the fix proposed by the analyzer developers and check that the issue

Table 2: Analyzer issues detected by Sherlock.

Issue ID	Program Analyzer	Issue Type	Issue Status	Seed LOCs
1	CBMC	Soundness	Fixed	57
2	CBMC	Soundness	Confirmed	1519
3	CBMC	Soundness	Confirmed	2820
4	CBMC	Precision	Confirmed	7
5	CBMC	Other	Confirmed	29
6	Clam	Soundness	Fixed	863
7	Clam	Precision	Fixed	24
8	Clam	Precision	Fixed	40
9	Clam	Soundness	Fixed	24
10	Clam	Soundness	Fixed	38
11	CPAchecker	Precision	Confirmed	42
12	ESBMC	Precision	Fixed	2747
13	ESBMC	Soundness	Fixed	2820
14	ESBMC	Soundness	Confirmed	2790
15	ESBMC	Soundness	Fixed	19
16	ESBMC	Soundness	Confirmed	13
17	KLEE	Soundness	Confirmed	41
18	KLEE	Soundness	Fixed	237
19	KLEE	Crash	Fixed	42
20	MOPSA	Soundness	Fixed	36
21	MOPSA	Precision	Confirmed	477
22	MOPSA	Soundness	Fixed	38
23	Symbiotic	Soundness	Reported	54
24	UAutomizer	Soundness	Confirmed	2820

no longer occurs. To account for the randomness in Sherlock, we repeat this experiment with 5 different random seeds.

We performed all experiments on a machine with two AMD EPYC 9474F CPUs @ 3.60GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm).

5.2 Results

We now discuss our findings for each research question.

RQ1: Issues detected by Sherlock. We completed the implementation of an initial version of Sherlock in the summer of 2023, and we reported the first issue in August. We incrementally added more transformations, analyzers, and seeds until April 2024, when we stopped testing. We used the issue-detection setup (see Sect. 5.1), and whenever an issue was detected, we either stopped testing the affected analyzer until a fix was provided or we manually de-duplicated any additional issues before reporting them. On average, we spent about 1–2 months testing each analyzer, except for Symbiotic, which we realized is no longer maintained.

Tab. 2 shows the list of unique analyzer issues detected by Sherlock. The first column provides an identifier for each issue and links to the (anonymized) bug report on GitHub/GitLab. The second column shows the program analyzer for which the issue was found. The next two columns indicate the type of issue (e.g., soundness or precision) and its status (e.g., reported, confirmed, or fixed). The last column shows the size of the seed program, in lines of code, that led to eventually detecting the issue.

In total, Sherlock detected 24 issues, 23 of which are confirmed by the analyzer developers and 13 are fixed. We found issues in all analyzers we tested, 16 of which are soundness issues and 6 are precision issues. As a by-product, we also found 2 issues (5 and 19) that are not related to soundness or precision. Issue 5 refers to CBMC's parser reporting an assertion violation on an incorrect line—the issue is revealed only for syntactically large expressions. Issue 19 is a crash in KLEE—our report triggered the developers to re-assess the effectiveness of a certain tool option and eventually remove it.

Although many of the confirmed issues have already been fixed, certain issues are non-trivial to fix. For instance, issue 2 was assigned high priority and triggered a long discussion among several CBMC developers. Issue 4 revealed an uncommon treatment of assertions in CBMC. More specifically, when analyzing executions through the successful branch of an assertion, the analyzer does not take into account that the asserted condition holds. Since this treatment may lead to false positives, a developer mentioned it might be revisited in the future. For now, the CBMC documentation has been updated to clarify the treatment of assertions.

RQ2: Description of detected issues. To better understand the detected issues, we discuss some of them in detail. Sherlock detected 16 soundness and 6 precision issues—in addition to the issues of Fig. 2 (1, 7, and 15), let us take a closer look at issues 9, 13, 18, 20, 22, and 8, all of which were fixed by the analyzer developers.

Fig. 3a shows the (minimized) program y that revealed soundness issue 9 in Clam. Given the non-deterministic integer `s`, the assertions on lines 3 and 5 should fail. Clam, however, missed the second assertion violation (line 5). This issue was caused by inconsistent handling of signed and unsigned integers in comparisons. Sherlock transformed program x to derive program y by replacing line 4 with 3; precisely, it weakened the assertion on line 4 (using WeakenAssert), thereby revealing the issue.

Fig. 3b revealed soundness issue 13 in ESBMC. The assertion on line 6, which compares pointers `str0` and `str1`, is unsoundly verified. Given this program x, Sherlock generated y by adding the assumption on line 2 (using AddAssume). For y, ESBMC correctly reports the assertion violation, and Sherlock detects the issue.

Soundness issue 18 was detected in KLEE with the program of Fig. 3c. Line 2 declares several non-deterministic, integer variables, and lines 3 and 5 specify relations among these using assumptions. Note that the assumptions do not imply the asserted condition on line 6, and the assertion may fail. Nevertheless, when KLEE is configured to verify this program, i.e., by turning off any unsound state-space pruning techniques, it unsoundly proves that the assertion holds. Given this program x, Sherlock generated y by adding the assumption on line 4. For y, KLEE soundly reports an assertion violation, thereby revealing the issue.

The KLEE developers found that the issue was caused by variable-name clashes when generating queries for Z3. All variable names were expected to be unique after appending a number suffix. However, when a name already ended with a number, uniqueness was no longer guaranteed, e.g., consider variable `v1` with the appended suffix `11` and variable `v11` with the appended suffix `1`. This resulted in solver queries that were semantically different than expected.

Fig. 3d revealed soundness issue 20 in MOPSA. For program x, shown in the figure, MOPSA unsoundly verified the assertion on line 6, which however fails for `a = 1` (and `x = 0.5`). Sherlock

```c
int main() {
  int S = ★;
  assert(S != 2U && S != 1U);
  // assert(S != 2U && S < 1U);
  assert(S > 2);
  return 0;
}
```

(a) Soundness issue in Clam.

```c
int main(){
  // assume(NULL != "a");
  char *str0 =
    (char *) 0xFFFFFFFFFFFFFFFF;
  char *str1 = "";
  assert(str0 <= str1);
}
```

(b) Soundness issue in ESBMC.

```c
void main() {
  int p1, ..., p15, cond = ★;
  assume(p12 > p14); assume(p6 > p3);
  // assume(p2 > 0);
  assume(p7 != 0); ... assume(0 > p4);
  assert(p2 > p11);
}
```

(c) Soundness issue in KLEE.

```c
int main() {
  int a = ★;
  assume(a <= 1);
  // assume(a <= 1 && a >= 1);
  double x = a / 2.0; ...
  assert(x <= 0.0); ...
}
```

(d) Soundness issue in MOPSA.

```c
int main() {
  int MAX = ★;
  char str1[MAX];
  assert(MAX >= 0);
}
```

(e) Soundness issue in MOPSA.

```c
void main() {
  N = ★;
  assume(N != -2147483648 &&
         N != 2147483647);
  for (int i = 0; i < N; i++)
    assert(N >= -1);
}
```

(f) Precision issue in Clam.

Figure 3: Additional soundness and precision issues detected by Sherlock and fixed by the analyzer developers.

generated y by strengthening the assumption on line 3 to the one on line 4, restricting the value of `a` to `1`. For y, MOPSA does report the assertion violation. This issue was likely caused by a copy-paste mistake. The fix involved changing a single line of code, where floating-point interval addition was replaced by multiplication in the definition of the backward multiplication operator. MOPSA also misses the assertion violation in Fig. 3e (issue 22). The issue occurs in the presence of undefined behavior, i.e., `MAX` is used to declare the length of array `str1` but may have a negative value.

Precision issue 8 was detected in Clam with the program shown in Fig. 3f. Since non-deterministic variable `N` is used as loop bound, the assertion on line 6 is only reachable with non-negative values of `N`; it can, therefore, not fail. Clam imprecisely reports an assertion violation due to the assumption on lines 3–4, which causes the analyzer to infer incorrect ranges for `N`. Without this assumption, Clam precisely verifies the assertion.

Overall, Sherlock detected issues in diverse components of program analyzers implementing a wide range of analysis techniques.

RQ3: Transformations. We now discuss which transformations helped to detect analyzer issues. For this and the remaining research questions, we use the issue-reproduction setup (see Sect. 5.1), which aims to re-find fixed issues with 5 independent random seeds. We only focus on fixed soundness and precision issues and omit by-product issue 19, which revealed a crash. We also omit issue 10 because, interestingly, Sherlock finds a (not yet fixed) precision issue for the given seed, and it is our minimization technique that reveals the soundness issue. Finally, we omit issue 12 because the fix involved disabling the buggy code altogether; we, thus, can no longer confirm whether a detected issue reveals the same problem.

Once Sherlock re-detected an issue, we inspected the sequence of stacked transformations that led to adding programs in the knowledge base and eventually finding the issue. The sequences, of course, varied across different, random seeds. We found that, *for all issues, transformations of each type (Strengthen, Weaken and Even) appeared in at least one sequence.* AddAssert and AddAssume constitute 60% of all transformations. While Sherlock chooses a transformation type uniformly (line 4 of Alg. 2), not all transformations may be applied at a particular location. For example, an assertion may always be added, whereas an assertion strengthening or weakening requires that an assertion exists at the location. The least frequent type of transformation is Even, making up only 2% of all transformations. This is because transformations of this type require that the location is instrumented with an assertion and that the analyzer verified this assertion. Another reason why transformations of type Even are rare is that the WeakenInvariant and AssumeInvariant transformations do not change the diversity profile of a program (see Sect. 3.4) and do not lead to adding the program to the knowledge base.

Interestingly however, these transformations do seem to contribute to revealing issues since they appear in 20% of the last stacked transformations in the sequences, i.e., the stacked transformations that led to detecting an issue. We observed that, despite changing the random seed, the last transformations for certain issues were always the same, indicating that they were necessary for detecting these issues. In particular, WeakenAssert or WeakenAssume were needed to detect issues 1 and 9, and WeakenInvariant or AssumeInvariant to detect issue 15.

RQ4: Sherlock vs. metamorphic testing. In this research question, we compare our approach to metamorphic testing. Note that we do not consider differential testing here for the following reasons. First, differential testing requires at least two analyzers to be applicable, whereas our approach can test a single analyzer in isolation. Second, it is challenging to differentially test two arbitrary analyzers, for instance, an abstract interpreter that is designed to be imprecise with a model checker that is designed to be unsound. This would inherently result in a large number of differences in the verification results [28]. Third, even when differentially testing two analyzers that implement the same technique, it is unclear which analyzer is at fault. None of these issues arise in our approach or in metamorphic testing.

As we discussed in Sect. 1, metamorphic testing (MT) of program analyzers typically applies the following workflow: query the analyzer on a program and get its response (i.e., the program is correct

Table 3: The transformations applied by MT and their respective oracles.

Type	Transformation	Oracle
Strengthen	AddAssume	$r_x = \checkmark \Rightarrow r_y = \checkmark$
	StrengthenAssume	$r_x = \checkmark \Rightarrow r_y = \checkmark$
Weaken	WeakenAssume	$r_x = \times \Rightarrow r_y = \times$
Even	WeakenInvariant	$r_y = \checkmark$
	AssumeInvariant	$r_y = \checkmark$

or incorrect); based on the response, transform the program such that the new expected response is known; query the analyzer again on the transformed program and get the new actual response; detect an issue if the new actual response does not match the expected one.

On the other hand, interrogation testing as instantiated in Sherlock uses more information than just the analyzer response to form queries; it additionally uses the verification results that the analyzer achieves for each query, that is, the assertions that are (un)verified. Moreover, it integrates a knowledge base that enables maintaining a history of analyzer queries. In this research question, we evaluate whether these two key innovations significantly increase the bug-finding effectiveness of Sherlock in comparison to MT. As there are no existing MT tools targeting these analyzers and types of issues, we implement MT in Sherlock. To create this baseline, we disable usage of the knowledge base and verification results.

Disabling the latter means that the detector may apply stacked transformations that solely rely on the analyzer response, and not its verification results. In other words, issues are detected based on the transformations and oracles shown in Tab. 3. In particular, transformations AddAssert, StrengthenAssert, and WeakenAssert may no longer be applied because we cannot define an oracle based only on the analyzer response. For instance, if an assertion is added to a correct program, we cannot predict the correctness of the resulting program. For AddAssume and StrengthenAssume, the oracle becomes "if program x is found to be correct, then program y must also be correct". If x is found to be incorrect and we add or strengthen an assumption, we cannot predict the correctness of the resulting program. Similarly, for WeakenAssume, the oracle becomes "if program x is found to be incorrect, then program y must also be incorrect". The oracles for transformations WeakenInvariant and AssumeInvariant remain the same, but these transformations may only be applied to correct programs.

The results of the MT and Sherlock comparison are shown in Tab. 4a—column MT+VR should be ignored for now. The table shows the average time (in minutes and over 5 runs with different random integer seeds) to re-find a fixed issue as well as for how many random seeds (out of 5) the issue is re-found. Recall that we only focus on fixed issues to ensure that a detected issue is indeed the one we intend to re-find (see Sect. 5.1). The average time in the table is computed by calculating the mean over the seeds that do detect an issue (i.e., by not counting timeouts). The best results per issue are shown in bold.

As shown in the table, *Sherlock is able to detect all 10 issues, whereas MT only finds 8. Moreover, Sherlock consistently detects, i.e., for all 5 random runs, 9 out of 10 issues, whereas MT only consistently detects 6. For issues that are consistently found by both MT and Sherlock, Sherlock achieves a speedup between 7x (for issue 9) and 906x (for issue 20).*

Sherlock did not consistently find issue 15. We observed that this issue was detected with the smallest seed program (across all 10 issues used in this experiment). Due to the small size of the seed program and our diversity criterion, the knowledge base could be filled with programs that are not effective in revealing the issue. This problem, which may only occur for small seeds, is alleviated when enabling TTL (see Sect. 3.4). In fact, when setting TTL = 250, Sherlock consistently detected the issue (across all 5 runs) within an average of 28 minutes. Note that, for the remaining issues, TTL with value 250 does not kick in, i.e., no programs are replaced in the knowledge base.

Tab. 4b shows the same results as Tab. 4a but for a time limit of two hours (instead of seven days). Sherlock still clearly outperforms MT. Specifically, Sherlock is able to detect 8 issues, whereas MT only finds 4. Moreover, Sherlock consistently detects 5 out of 8 issues, whereas MT only consistently detects 3. For issues that are consistently found by both MT and Sherlock, Sherlock achieves a speedup between 7x and 8x.

RQ5: Knowledge base. In this research question, we extend MT with verification results (VR), to obtain MT+VR, and compare it with Sherlock. In other words, MT+VR may apply the same transformations as Sherlock, i.e., those of Tab. 1, but it may not use a knowledge base. The results are shown Tab. 4a, when comparing columns MT+VR and Sherlock.

As shown in the table, *Sherlock is able to detect all 10 issues, whereas MT+VR only finds 9. Moreover, Sherlock consistently detects, i.e., for all 5 random runs, 9 out of 10 issues (when not enabling TTL), whereas MT+VR only consistently detects 7. For issues that are consistently found by both MT+VR and Sherlock, Sherlock achieves a speedup between 7x (for issue 7) and 20x (for issue 20).*

Observe that issue 13 could not be detected by MT, but it is detected when enabling the use of verification results in MT+VR. Moreover, note that issue 18 cannot be detected without the use of the knowledge base. This issue was found in KLEE, which often timed out when being tested with MT or MT+VR. In contrast, the knowledge base in Sherlock helps to filter and retain "good" programs, which can be handled by KLEE without a timeout. This indicates that, over time, the queries become more and more targeted to the capabilities of the analyzer under test.

In Tab. 4b (for the 2-hour time limit), Sherlock is able to detect 8 issues, whereas MT+VR only finds 6. Moreover, Sherlock consistently detects 5 out of 8 issues (when not enabling TTL), whereas MT+VR only consistently detects 4. For issues that are consistently found by both MT+VR and Sherlock, Sherlock achieves a speedup between 7x and 20x.

5.3 Threats to Validity

Our experimental results depend on the program analyzers under test, the seed programs, and randomness in the applied transformations. To address the first potential threat, we diversified the set of analyzers we tested. In particular, we selected mature analyzers that implement conceptually different analysis techniques. Similarly, we

Issue ID	MT		MT+VR		Sherlock	
	Time	Seeds	Time	Seeds	Time	Seeds
1	8814	1	3567	3	**521**	**5**
6	8576	1	1779	5	**97**	**5**
7	32	5	26	5	**4**	**5**
8	1117	5	78	5	**9**	**5**
9	21	5	23	5	**3**	**5**
13	-	0	4351	1	**4142**	**5**
15	49	5	**8**	**5**	< 1	1
18	-	0	-	0	**1369**	**5**
20	1811	5	39	5	**2**	**5**
22	738	5	161	5	**20**	**5**

(a) 7-day time limit

Issue ID	MT		MT+VR		Sherlock	
	Time	Seeds	Time	Seeds	Time	Seeds
1	-	0	-	0	**15**	**1**
6	-	0	-	0	**88**	**4**
7	32	5	26	5	**4**	**5**
8	-	0	60	4	**9**	**5**
9	21	5	23	5	**3**	**5**
13	-	0	-	0	-	0
15	49	5	**8**	**5**	< 1	1
18	-	0	-	0	-	0
20	-	0	39	5	**2**	**5**
22	10	1	15	3	**20**	**5**

(b) 2-hour time limit

Figure 4: Average time to detect a fixed issue for MT, MT+VR, and Sherlock (7-day time limit for (a) and 2-hour time limit for (b)). Time is displayed in minutes, averaged across 5 runs with different random integer seeds.

selected the seed programs from popular open-source repositories or we generated them automatically. Lastly, to limit the effect of randomness in our results, we ran Sherlock as well as the MT and MT+VR baselines multiple times using independent random seeds.

6 Related Work

The most closely related work aims to find bugs in program analyzers or their components, for instance, abstract domains [9, 33], constraint solvers [31, 35, 41, 42], or Datalog engines [30, 32].

In Sect. 1, we have described four high-level approaches for analyzer testing, including their limitations. These approaches are based on (1) program generation (e.g., [8, 23]), (2) specification-based testing (e.g., [9, 33, 40]), (3) differential testing (e.g., [18, 25, 28, 35, 41]), and (4) metamorphic testing (e.g., [20, 30–32, 34, 42, 44–46]). In contrast, interrogation testing introduces a novel testing framework that incorporates ideas from metamorphic testing [13, 36] (to create powerful test oracles) and greybox fuzzing [3] (to generate more diverse queries using a knowledge base that captures more information than just the analyzer response).

There are two interesting connections with recent work we emphasize next. Feedback-directed metamorphic testing (FDMT) [39] extends metamorphic testing to dynamically adjust how metamorphic relations and source test cases (i.e., test cases that are transformed by metamorphic relations) are selected. FDMT and interrogation testing both aim to improve over random selection of source test cases but use very different approaches. For instance, interrogation testing selects diverse tests based on the *output* of the tested analyzer, while FDMT uses partition testing that typically focuses on the test *input*.

In contrast to both interrogation testing and FDMT, GrayC [18] aims to improve the selection of source test cases that are subsequently used for *differential testing* of program analyzers (and compilers). GrayC builds a corpus of interesting test cases by collecting code-coverage feedback during compilation. In contrast, our diversity metric uses the verification results of the tested analyzer.

Compiler testing [12] is another related area given that compilers often incorporate fast and scalable analysis components, such as data-flow analyses. These analyses may introduce compilation failures, for instance, by wrongly optimizing the program based on results from an incorrect data-flow computation. However, finding bugs in the analysis components is essentially a by-product, not the main objective. In contrast, interrogation testing aims to find issues in general-purpose program analyzers that use a much wider range of analysis techniques, such as abstract interpretation [17], bounded model checking [7], and symbolic execution [11, 26].

7 Conclusion

We have presented interrogation testing, a novel systematic testing framework for detecting soundness and precision issues in program analyzers. Our approach introduces two key innovations that drastically boost its bug-finding effectiveness (as shown in our experimental evaluation): (1) incorporating more information from analyzer queries to obtain more powerful oracles, and (2) introducing a knowledge base maintaining a history of diverse queries.

We have also described a concrete instantiation of this general framework for program analyzers that reason about reachability; these include a wide range of analyzers, such as model checkers, abstract interpreters, and symbolic-execution engines. Sherlock, the corresponding tool, has already revealed 24 unique issues (16 soundness and 6 precision issues), 23 of which were confirmed by the analyzer developers. Sherlock found issues in all 8 analyzers we tested, and soundness issues in 7 of them.

In future work, we plan to explore new instantiations of interrogation testing for other classes of analyzers (e.g., taint 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.]. C-Reduce, A C and C++ Program Reducer. https://github.com/csmith-project/creduce.
[2] [n. d.]. Goblint. https://github.com/goblint/analyzer.
[3] [n. d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[4] Dirk Beyer. 2012. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org.

[5] Dirk Beyer. 2019. Competition on Software Testing (Test-COMP). https://test-comp.sosy-lab.org.

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

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *TACAS (LNCS, Vol. 1579)*. Springer, 193–207.

[8] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.

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

[10] 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.

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *CACM* 56 (2013), 82–90. Issue 2.

[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.

[13] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.

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

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

[16] 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.

[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] 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.

[19] Arie Gurfinkel and Jorge A. Navas. 2021. Abstract Interpretation of LLVM with a Region-Based Memory Model. In *VSTTE (LNCS, Vol. 13124)*. Springer, 122–144.

[20] 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.

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

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

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

[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] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19 (1976), 385–394. Issue 7.

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*. ACM, 207–220.

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

[29] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52 (2009), 107–115. Issue 7.

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

[31] 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.

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

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

[34] 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.

[35] 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.

[36] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *TSE* 42 (2016), 805–824. Issue 9.

[37] 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.

[38] 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.

[39] Chang-Ai Sun, Hepeng Dai, Huai Liu, and Tsong Yueh Chen. 2023. Feedback-Directed Metamorphic Testing. *TOSEM* 32 (2023), 20:1–20:34. Issue 1.

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

[41] 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.

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

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

[44] 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.

[45] 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.

[46] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *CoRR* abs/2402.14366 (2024).