# Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles

WEIGANG HE, East China Normal University, China and University of Technology Sydney, Australia
PENG DI, Ant Group, China
MENGLI MING, East China Normal University, China
CHENGYU ZHANG, ETH Zurich, Switzerland
TING SU*, East China Normal University, China
SHIJIE LI, Ant Group, China
YULEI SUI, University of New South Wales, Australia

Static analyzers are playing crucial roles in helping find programming mistakes and security vulnerabilities. The correctness of their analysis results is crucial for the usability in practice. Otherwise, the potential defects in these analyzers (*e.g.*, implementation errors, improper design choices) could affect the soundness (leading to false negatives) and precision (leading to false positives). However, finding the defects in off-the-shelf static analyzers is challenging because these analyzers usually lack clear and complete specifications, and the results of different analyzers may differ. To this end, this paper designs two novel types of automated oracles to find defects in static analyzers with randomly generated programs. The first oracle is constructed by using dynamic program executions and the second one leverages the inferred static analysis results. We applied these two oracles on three state-of-the-art static analyzers: Clang Static Analyzer (CSA), GCC Static Analyzer (GSA), and Pinpoint. We found 38 unique defects in these analyzers, 28 of which have been confirmed or fixed by the developers. We conducted a case study on these found defects followed by several insights and lessons learned for improving and better understanding static analyzers. We have made all the artifacts publicly available at https://github.com/Geoffrey1014/SA_Bugs for replication and benefit the community.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Static analyzers, defects, automated oracles construction

**ACM Reference Format:**
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. *Proc. ACM Softw. Eng.* 1, FSE, Article 74 (July 2024), 23 pages. https://doi.org/10.1145/3660781

## 1 INTRODUCTION

Static analysis tools (*static analyzers* for short), *e.g.*, [9, 13, 22, 31], are widely used for finding program bugs like common programming mistakes and security vulnerabilities. When integrated

*Corresponding author.

Authors' addresses: Weigang He, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China and University of Technology Sydney, Australia, geoffreyhe2@gmail.com; Peng Di, Ant Group, China, dipeng.dp@antgroup.com; Mengli Ming, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China, dale.mengli.ming@proton.me; Chengyu Zhang, ETH Zurich, Switzerland, dale.chengyu.zhang@gmail.com; Ting Su, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China, tsu@sei.ecnu.edu.cn; Shijie Li, Ant Group, China, lishijie.lsj@antgroup.com; Yulei Sui, University of New South Wales, Australia, y.sui@unsw.edu.au.


ACM 2994-970X/2024/7-ART74
https://doi.org/10.1145/3660781

Listing 1. The program revealing defect #1 in GSA through the dynamic oracle.

```
1  void main() {
2    int e = 42, g = 0;
3    int *f = &e;
4    int *h[2][1];
5    h[1][0] = f;
6    if (g == (h[1][0]))
7      unsigned int *i = 0;
8    printf("NPD_FLAG:%d\n", *f); // GSA gives a warning of NPD
9  }
```

into the software development process, these analyzers can help uncover program bugs at the early stage (after compilation and before testing). Therefore, whether these analyzers could give correct analysis results, is crucial for their usability in practice [30]. Otherwise, the potential defects in these analyzers could affect the soundness (*i.e.*, missing true bugs and thus inducing false negatives) and precision (*i.e.*, reporting spurious warnings [14] and thus inducing false positives).

However, finding the defects in off-the-shelf static analyzers is challenging. One key difficulty is the oracle problem, *i.e.*, how to verify whether a static analyzer gives the correct analysis results on a given program. On one hand, since static analyzers usually do not have clear and complete specifications, they are difficult to be formally verified like compilers [4]. On the other hand, since static analyzers usually use different strategies to trade-off the soundness and precision, their analysis results are difficult to be validated by differential testing [22]. To our knowledge, most of prior work only tests specific static analyses (*e.g.*, value analysis [10], alias analysis [47], data-flow analysis [37]) rather than off-the-shelf analzyers, which we will discuss in Section 6.

To this end, in this paper we introduce two novel types of oracles to help find defects in static analyzers. One of our observations is that that many static analyzers provide *checkers* [38, 42] to find specific bugs (*e.g.*, *null pointer dereference* (NPD)). Thus, our insight is to leverage such checkers as the handler to test the static analyzers. Ideally, the results inferred by a static analyzer's checker on a program should be consistent with the results obtained by running the program. For example, if an NPD is triggered on some program path by running a given program, the static analyzer's NPD checker should report this NPD as well. Otherwise, a soundness bug (*i.e.*, a false negative) is found. If no NPD is triggered by running the program, the analyzer's NPD checker should not report any NPD on the path. Otherwise, a precision bug (*i.e.*, a false positive) is found. We term this type of oracle which refers to dynamic program execution as *the dynamic oracle*.

To realize this dynamic oracle, in the case of finding false negatives, we inject a ground-truth bug such as an NPD into a given program and dynamically execute the program to verify the NPD. In this case, if the static analyzer's NPD checker fails to report this injected bug, we find a false negative. In the case of finding false positives, we dynamically execute a given program to ensure the absence of a specific bug like an NPD. In this case, if the static analyzer's NPD checker reports some NPD warnings, we find a false positive. Note that this dynamic oracle is not limited to the NPD checker but could be applied to any checker which targets runtime errors (discussed in Section 5). By using this dynamic oracle, the code in Listing 1 exposes a defect in GCC Static Analyzer (GSA) [13]. GSA reports a spurious warning of NPD at line 8 (because variable `f` is not `null`). This defect[1] has been quickly fixed by the developers after we reported.

Our another observation is that many static analyzers provide debug checks [39, 43] to evaluate program states. Specifically, such debug checks takes as input a boolean expression (representing some program state, *e.g.*, " `x==5`") at a given program location, and return `TRUE`, `FALSE` or `UNKNOWN` if

[1] *https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107345*

Listing 2. The program revealing defect #3 in GSA through the static oracle.

```
1  void foo(){
2    int *b[1] = {};
3    int **c = &b[0];
4    if (c == &b[0]){
5      __analyzer_eval((c+0) == (&b[0]+0)); // TRUE
6      __analyzer_eval((c+1) == (&b[0]+1)); // FALSE
7    }
8  }
```

the expression is evaluated to a non-zero value, a zero or `null` value, or is not sufficiently constrained. Thus, our insight is to generate equivalent but different representations of such boolean expressions to stress test the static analysis results. Ideally, the truth values of equivalent boolean expressions should be identical. Otherwise, a potential defect is likely found. We term this type of oracle which only refers to the static analysis results on program states as *the static oracle*.

To realize this static oracle, we select the conditional expression of some conditional statement in a given program as the target boolean expression. Note that this boolean expression is evaluated to be `TRUE` by construction at the entry of the true branch of its conditional statement. Therefore, we can instrument debug checks taking equivalent boolean expressions at the entry of same true branch and to evaluate the truth values. If some debug check returns `FALSE`, we likely find a potential defect of the analyzer. Specifically, we design two mutation strategies based on the idea of metamorphic testing [7] (discussed in Section 3.3) to generate equivalent boolean expressions but with different representations. By using this oracle, the code in Listing 2 exposes a defect in GSA. In the true branch of `if (c == &b[0])`, GSA should evaluate both `(c + 1) == (&b[0] + 1)` and `(c + 0) == (&b[0] + 0)`, the two equivalent boolean expressions of `c == &b[0]`, to be `TRUE`. However, GSA evaluates `(c + 1) == (&b[0] + 1)` to be `FALSE`. This defect has been confirmed by the developers[2]. Section 5 discusses that the static and dynamic oracles are complementary in finding defects.

We applied the two types of oracles to test three state-of-the-art static analyzers, *i.e.*, Clang Static Analyzer (CSA) [18], GCC Static Analyzer (GSA) [13], and Pinpoint [31]. CSA and GSA are open-sourced and well-maintained by a large community, both of which adopt symbolic execution [5]. Pinpoint is a commercial analyzer designed for enterprise-level large software based on sparse value flow analysis [8, 33, 34]. Pinpoint has been integrated into the development process of tens of FinTech enterprises to improve software quality. We used Csmith [48] to generate random input programs for testing these analyzers. When a potential defect is found, we use the program reduction technique [29] to obtain a concise and reproducible test program for bug reporting.

In the testing campaign, we found 38 unique defects in these three state-of-the-art static analyzers, 28 of which have been confirmed and four have been fixed by the developers. Among these 28 confirmed bugs, 24 of them are new, previously-unknown defects, while the other four defects were also independently reported by others. Additionally, three defect-triggering programs have been integrated into the regression tests of GSA.

To understand these defects, we carefully classify them into 8 major groups from low-level to high-level issues (*e.g.*, *implementation errors*, *defects in heuristics*, *improper design choices*) based on the discussions with developers, fixing patches, and our knowledge on static analysis. We discuss and illustrate each type of defect with examples. We believe the results of our work is beneficial to the static analyzers' developers for improving their tools as well as the analyzers' users to better understand the tool limitations. To benefit the community, we have made the

[2] *https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109199*

Table 1. The Three Studied Static Analyzers

| Static Analyzer | Intermediate Representation | Inner Workings | Supported Language |
|---|---|---|---|
| CSA | Clang AST | Symbolic Execution | C, C++, Object-C |
| GSA | Gimple IR | Symbolic Execution | C, C++, D |
| Pinpoint | LLVM IR | Sparse Value Flow Analysis | C, C++, JAVA |

artifacts (including our testing tool and the defect-triggering programs) publicly available at https://github.com/Geoffrey1014/SA_Bugs.

In summary, the main contributions of our work include:

- We propose two novel types of test oracles to help find defects in off-the-shelf static analyzers.
- We apply the proposed oracles on three static analyzers and found 38 unique defects (28 defects have been confirmed and four have been fixed by the developers).
- We conduct a case study on the found defects and distill several lessons learned on improving and better understanding static analyzers.

## 2 BACKGROUND

### 2.1 Static Analyzers

In this section, we provide the necessary background on static analyzers. The main objective of static analyzers is to compute program facts that should hold at a given program location. The static analyzers usually accept an input program and use different static analyses (*e.g.*, data-flow analysis, symbolic execution) to derive an abstract representation that encapsulates the program's facts. These facts can be used to find program bugs like security vulnerabilities.

Table 1 lists the three static analyzer studied in our work, *i.e.*, Clang Static Analyzer [9], GCC Static Analyzer [13], and Pinpoint [31]. We selected these analyzers because they are (1) widely-used in practice, and (2) representatives of static analyzers using different static analyses.

**Clang Static Analyzer (CSA)** [9] provides a top-tier static analysis platform for scrutinizing programs written in C, C++, and Objective-C. It has undergone significant development since 2008, evolving from basic syntactic checkers to advanced tools capable of identifying complex issues by analyzing code semantics. CSA utilizes symbolic execution with a source code simulator that traces out possible paths of execution. It runs the inline method to perform inter-procedural analysis (IPA). CSA can find program bugs such as null-pointer dereference (NPD) and out of bounds (OOB). The analyzer includes two major parts: symbolic execution of a given program and specially designed checkers that subscribe to the events they find relevant during analysis .

**GCC Static Analyzer (GSA)** [13] has been developed since 2019 with the purpose of scanning the Linux kernel. Like CSA, GSA employs symbolic execution over GCC's IR (Gimple). Implemented as an optimization pass, GSA can combine the IRs from different compilation units in various languages and find program bugs across language boundaries during link-time optimization.

**Pinpoint** [31] is a commercial static analyzer developed by Ant Group since 2018. The tool implements the sparse value flow analysis, which is categorized as a "layered" approach. By first finding local data dependence and delaying costly inter-procedural data dependence analysis, Pinpoint can reduce the cost of conducting high-precision points-to analysis. This capability enables Pinpoint to easily scale bug checkers to millions of lines of code. Its commercial version is now widely used in FinTech businesses [23, 51, 52].

All these three static analyzers include a number of checkers [38, 42] for finding specific program bugs, and report warnings for user inspection. For example, null-pointer dereference (NPD) and out of bounds (OOB), and memory leaks are the common targets of these checkers.

Listing 3. The error trace generated by GSA of the program revealing defect #1.

```
<source>:8:1: warning: dereference of NULL 'f'
[CWE-476]
8 | printf("NPD_FLAG: %d\n ", *f);
| ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'main': events 1-4
|   6 |  if (g == (h[1][0])) {
|     |     ^
|     |     |
|     |     (1) following 'true' branch...
|   7 |    int *i = 0;}
|     |         ~
|     |         |
|     |         (2) ...to here
|     |         (3) '&e' is NULL
|   8 | printf("NPD_FLAG: %d\n ", *f);
|     | ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|     | |
|     | (4) dereference of NULL 'f'
```

Listing 4. Evaluation results of GSA's debug check `__analyzer_eval` for defect #3.

```
<source>: In function 'main':
<source>:5:17: warning: TRUE
5   |   __analyzer_eval((c + 0) == (&b[0] + 0));
|         ^~~~~~~~~~~~~~~~~~~~~~~~
<source>:6:17: warning: FALSE
6   |   __analyzer_eval((c + 1) == (&b[0] + 1));
|       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

## 2.2 Error Traces and Debug Checks

The static analyzers usually report the warnings (*i.e.*, the found bugs) to users in the form of an error trace (or named as a *diagnostic path*) to facilitate bug diagnosing. This trace is composed of a finite sequence of steps that lead to the reported bug. Each step is represented by a tuple $\langle location, message \rangle$, where *location* denotes the file, line, and column associated with the step, and *message* specifies the condition inferred by the analyzer should hold to reach the bug. Listing 3 displays the trace reported by GSA when analyzing the code in Listing 1. The trace gives the steps leading to the NPD at line 8, and the conditions inferred by the analyzer should hold at each step.

The static analyzers like GSA and CSA provide debug checks [39, 43] to evaluate specific program states. For example, GSA and CSA provide the `__analyzer_eval` [43] and `clang_analyzer_eval` [39] functions for inspecting the program states at a given program location. At a given program location, these functions take as input a boolean expression (representing some program state), and return three possible values: `TRUE`, `FALSE`, or `UNKNOWN`. These three values correspond to the situation when the static analyzer decides the inspected program state is hold, not hold or unknown. Pinpoint [31] does not export such debug checks. Thus, we exclude Pinpoint from the static oracle evaluation. Listing 4 showcases that GSA evaluates lines 5 and 6 of the code in Listing 2 by using its debug check `__analyzer_eval`. Through this check, GSA infers that `(c + 0) == (&b[0] + 0)` is TRUE, while `(c + 1) == (&b[0] + 1)` is FALSE at the given program location.

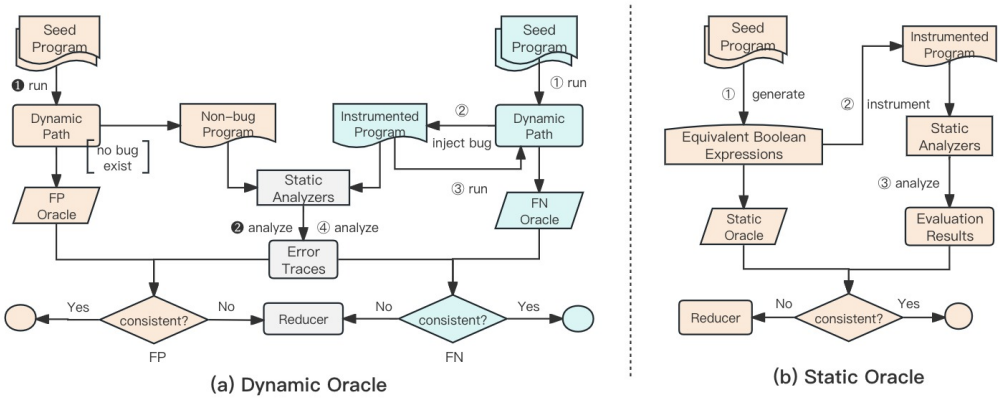


Fig. 1. Workflows of the two types of automated oracles for finding defects in static analyzers.

## 3 METHODOLOGY

In this section, we introduce the methodology of our two types of oracles. First, we explain why we use randomly generated programs instead of real-world programs to create oracles in Section 3.1. In Sections 3.2 and 3.3, we introduce the workflows of our two types of oracles. At last, we explain how to reduce the defect-triggering programs in Section 3.4.

### 3.1 Generating Random Programs

We choose Csmith to generate random programs instead of real-world ones for testing static analyzers because of the following advantages:

(1) Csmith-generated programs incorporate diverse language features. These features can be enabled or disabled through command-line options. This facilitates us to disable certain language features that are not well modeled by static analyzers, *e.g.*, *global variables* and *bitfields*.
(2) Csmith-generated programs are closed, which does not require external inputs and executes one single program path. This characteristic ensures that the program's execution path is deterministic, which eases bug injection and validation for the dynamic oracle.
(3) Csmith-generated C programs are guaranteed, through Csmith's inner static analysis, to be free of undefined or unspecified behaviors. This feature is crucial because, otherwise during program compilation and analysis, the compiler and static analyzers may exploit undefined or unspecified behavior in the program in different ways, and thus affects testing static analyzers.

### 3.2 Dynamic Oracle

The dynamic oracle is designed based on our insight that the results inferred by a static analyzer's checker on a program should be consistent with the results obtained by running the program. Thus, we can use a Csmith-generated random program as a test input *and* its dynamic execution results (*i.e.*, the existence of a runtime error on some program path) as the test oracle to validate the static analyzer. In our study, we selected the NPD checker as the handler to test static analyzers. We selected the NPD checker because it is one of the most common ones supported by most static analyzers, including GSA, CSA and Pinpoint. Note that the dynamic oracle can be applied to any checker which targets runtime errors (*e.g.*, NPDs, OOBs, and division by zero). Figure 1(a) shows the workflow of constructing the dynamic oracle, which we explain as follows. Note that the left part (in yellow) of Figure 1(a) indicates the dynamic oralce for finding false positives, while the right part (in green) of Figure 1(a) indicates the dynamic oralce for finding false negatives.

**Algorithm 1** Construct the Static Oracles

**Require:** input program $P$
1: $e, c \leftarrow GetCandidateExpression(P)$; // Pick candidate expression $e$ of a conditional statement $c$
2: $e' \leftarrow Mutate(e)$; // Mutate the boolean expression $e$
3: $s \leftarrow Wrap(e')$; // Wrap $e'$ with a debug check
4: $P' \leftarrow InsertAtLocation(P, s, c)$; // Instrument $s$ at the true branch of $c$ into $P$
**Ensure:** program variant $P'$

**To find false positives of a static analyzer**, the program generated by Csmith is compiled and executed to confirm the absence of an NPD (the step ❶ in Figure 1(a)). Only those non-bug programs are qualified to be analyzed by the static analyzer (the step ❷ in Figure 1(a)). If the analyzer reports some NPD warnings (which are presented in the form of error traces), we likely find some false positives. However, we are only interested in such false positives: (1) the reported NPD is located on the dynamic program path, and (2) the reported error trace leading to the NPD is started from function `main`[3]. We find the valid false positives satisfying the preceding two conditions by checking the consistency between the dynamic program path (used as the FP oracle) and the error traces. We do not require the error trace to exactly match the dynamic path, because the error trace may not always be precise [22].

**To find false negatives of a static analyzer**, we inject the bugs of NPDs into the Csmith-generated programs. To achieve this, we first run a seed program to obtain the dynamic execution path (the step ① in Figure 1(a)), randomly assign `null` to some local variables of `pointer` type on the dynamic path to inject NPDs (the step ② in Figure 1(a)), and rerun the instrumented program to confirm that some NPDs can be triggered on the dynamic path (the step ③ in Figure 1(a)). Only those NPD-triggering programs are qualified to be analyzed by the static analyzer (the step ④ in Figure 1(a)). If the analyzer fails to report the injected NPDs, we likely find false negatives. Note that the analyzer successfully finds the injected NPDs *only when* the following two constraints are satisfied: 1) the NPDs are reported on the program locations where they are injected, and 2) the error traces given by the static analyzer should start from function `main` . If one or both constraints are not satisfied, the analyzer fails to find the NPD and thus a false negative is found. We filter the valid false negatives satisfying the preceding constraints by checking the consistency between the dynamic program path (used as the FN oracle) and the error traces.

### 3.3 Static Oracle

The static oracle is designed based on the insight that the truth values of equivalent boolean expressions should be identical when evaluated by the debug checks at a given program location. Figure 1(b) shows the workflow of constructing the static oracle. We select the conditional expression of some conditional statement in a given program as the target boolean expression. This boolean expression is `TRUE` by construction at the entry of the true branch of its conditional statement. Based on this boolean expression, we generate equivalent boolean expressions with different representations (the step ① in Figure 1(b)). Next, we instrument the debug checks taking the generated equivalent boolean expressions at the entry of same true branch (*e.g.*, lines 5 and 6 show in Listing 2, the step ② in Figure 1(b)), and run the static analyzer to obtain the evaluation results of boolean expressions (the step ③ in Figure 1(b)). If the evaluation results of these equivalent boolean expressions are not consistent with the truth value of target boolean expression (*i.e.*, some debug check returns `FALSE`), we likely find a potential defect of the analyzer (as shown in Listing 4).

[3]Note that the top-level function for static analysis is not always function `main`. For example, the static analyzer may report an NPD bug whose error trace starts from some function `foo` in the program (`foo` may not be called by `main`)

Table 2. Mutation strategies to generate equivalent boolean expressions for the static oracle. $M$ and $N$ are in the range of {0,1,2,3}, and $M <= N$. $MIN$ and $MAX$ denote the minimal and maximal values depending on variable types, respectively.

| Operator | Expr | Tautological Transformation | Partial Ordering Transformation |
|---|---|---|---|
| && | a && b | $(b\&\&a) == true; a == true; b == true; !a == false; !b == false;$ $!(b\&\&a) == true; (!a\|\|!b) == false; (!b\|\|!a) == false; (!a\&\&!b) == false; (!b\&\&!a) == false;$ $(b\|\|!a) == true; (a\|\|!b) == true; !b\|\|a == true; (!b\&\&a) == false; (a\&\&!b) == false;$ $!(a\&\&b) == true; (!a\|\|b) == true; (b\&\&!a) == false; (!a\&\&b) == false;$ | – |
\|\| | a\|\|b | $(b\|\|a) == true; a == true\|\|b == true; !(a\|\|b) == false;$ $(!a\&\&!b) == false; (!b\&\&!a) == false; !(b\|\|a) == false;$ | – |
< | $a < b$ | $(b > a) == true; !(a < b) == false; (a >= b) == false; (a > b) == false; (a == b) == false;$ | $MIN <= a + M < b + N <= MAX$ |
> | $a > b$ | $(b < a) == true; !(a > b) == false; (a <= b) == false; (a < b) == false; (a == b) == false;$ | $MAX >= a + N > b + M >= MIN$ |
<= | $a <= b$ | $(b >= a) == true; !(a <= b) == false; (a > b) == false;$ | $MIN <= a + M <= b + N <= MAX$ |
>= | $a >= b$ | $(b <= a) == true; !(a >= b) == false; (a < b) == false;$ | $MAX >= a + N >= b + M >= MIN$ |
== | $a == b$ | $(b + M == a + M) == true; (a - M == b - M) == true; (a * M == b * M) == true;$ | $MIN <= a + M < b + N <= MAX$ |

Algorithm 1 explains constructing the static oracle. Given an input program $P$, we pick a boolean expression $e$ (*e.g.*, `c == &b[0]` at line 4 in Listing 2) from a conditional statement $c$ as the target. To ease generating equivalent boolean expressions, we require $e$ to be free of side effect on the program state. For example, if $e$ does some computation (*e.g.*, containing the += and -= operators), we give up $e$ and select others. When we have a qualified $e$, we use the two mutation strategies in Table 2 to generate equivalent $e'$ from $e$. Next, we wrap the mutated expression $e'$ with the debug check, *e.g.*, `__analyzer_eval()`, to get a new statement $s$. Finally, we insert $s$ at the entry of the true branch of the conditional statement $c$, and obtain a mutated program $P'$.

The generated equivalent boolean expression should be evaluated as `TRUE`. When the analyzer analyzes $P'$, if some debug check returns `FALSE`, we likely find a defect. If some debug check returns `UNKNOWN`, it may also indicate a defect. However, in practice, the static analyzers give lots of `UNKNOWN`, and thus we focus on the return value `FALSE`.

Table 2 shows the two mutation strategies, *i.e.*, (1) *tautological transformation*, and (2) *partial ordering transformation*, in terms of logical and relational operators (see Column "Operator"). These two mutation strategies are designed based on the idea of metamorphic testing [7]. Column "Expr" gives different forms of target boolean expressions, where `a` and `b` represent expressions or variables. For instance, if we have the boolean expression `a && b`, we can generate an equivalent boolean expression `!a || !b == false` by *tautological transformation*. On the other hand, we can generate an equivalent expression by performing arithmetic operations such as addition, subtraction, and multiplication on the relational expressions. For example, if we have the boolean expression `a == b`, we can generate an equivalent boolean expression, `a + 1 > b && a < INT_MAX` (`a` is an integer), by *partial ordering transformation*. Note that during the partial ordering transformation, we need to ascertain the types of left and right subexpressions. Fortunately, due to the strong typing of the C programming language, the types can be obtained during syntactic parsing. For logical expressions, we can only apply the tautological transformation, while we can use both tautological and partial ordering transformations for relational expressions.

One advantage of this static approach is that we do not need to use a checker (*e.g.*, NPD checker) as the handler. One checker may only utilizes limited program facts computed by the static analyzer because it targets specific runtime error. The static oracle may test different program facts with the help of debug checks. In addition, the static oracle may reveal the defects of static analyzers when handling dead code, while the dynamic oracle cannot do that. On the other hand, the disadvantage is that static analyzers without debug checks cannot be tested by the static oracle.

### 3.4 Reducing Defect-triggering Programs

When we find a potential defect of a static analyzer, we need to report it to the developer for confirmation. To help developers diagnose the defect, we need to provide a concise and reproducible defect-triggering program. Unfortunately, the programs generated by Csmith are difficult for humans to read and reduce. Therefore, we use C-Reduce [29] to reduce the program.

Specifically, C-Reduce requires a script to decide whether the program during reduction always hold the desired properties. In our setting, the script should ensure the behaviors of a found defect always hold during reduction. However, undefined behaviors (UB) may be introduced when reducing the program. In order to avoid UBs, when reducing the defect-triggering programs generated by the dynamic oracle, we include CompCert [20] into the C-Reduce's script to interpret the execution of the reduced program to sanitize UBs. On the other hand, when reducing the defect-triggering programs generated by the static oracle, we use some checkers in the static analyzer to sanitize UBs. After we obtain the final reduced program, we use CompCert again to determine whether the program contains UBs.

## 4 CASE STUDIES

In this section, we present our experience of applying the two types of oracles to find defects in CSA, GSA, and Pinpoint. We classify and discuss the defects found by our oracles.

### 4.1 Experimental Setup

We detail the experimental setups used in our testing campaign. We used Csmith 2.3.0 to generate seed programs with the options "`-max-pointer-depth 2 -no-bitfields -no-global-variables`". These three options limit the indirect depth of pointers to 2 and disable full-bitfields structs and global variables[4] when generating the seed programs, respectively. During the testing campaign, we tested the latest versions of CSA (Clang commit `0c0681b`), GSA (GCC commit `8c8ca87`), and Pinpoint (v2.7) at the time of our study. CSA, GSA, and Pinpoint were run with their default settings, respectively: CSA uses "`clang -analyze`", GSA uses "`gcc -fanalyzer`", and Pinpoint uses "`pp-check`". In the testing scenario of *dynamic oracle*, we use the NPD checkers of these three analyzers as the handler to find defects. In Section 5, we discuss the results of using other checkers like OOB checkers. In the testing scenario of *static oracle*, CSA was additionally given the option "`-analyzer-checker=debug.ExprInspection`" to enable the debug checks, while GSA by default enables the debug checks. Pinpoint does not support debug checks and thus was not tested.

We implemented the testing framework for static analyzers in `Python` and `C++`. Specifically, the fuzzing and reduction scripts contain about 2000 lines of `Python` code. We used C-Reduce (v2.11.0) as the reduction tool. We used Libtooling [44] for generating and instrumenting the static oracle into the seed programs. This module contains about 1100 lines of C++ code. Additionally, considering GSA, implemented as an IPA pass in GCC, could be affected by compiler optimization, we performed a separate testing campaign for GSA with the optimization levels of `-O0`, `-O1`, and `-O2`, respectively. Since CSA and Pinpoint are implemented as independent analysis tools, and thus are not affected by compiler optimization.

Our experiment was conducted on a 2.2 GHz Intel dual-CPU 64-core machine with 128GB memory running 64-bit Ubuntu 20.04.3 LTS. We started our testing campaign in September 2022 and continued until July 2023. During this time duration, the testing process was *iteratively* conducted:

[4]After a preliminary testing campaign, we find that the three static analyzers report too many duplicated false positives due to their weaknesses of handling bitfields and global variables. Thus, we added the two options `-no-bitfields` and `-no-global-variables`. On the other hand, the option `-max-pointer-depth 2` balances the complexity and readability of the generated seed programs.

Table 3. Results of the found defects in the studied static analyzers

| Static Analyzers | Dynamic Oracle | Static Oracle | Total |
|---|---|---|---|
| CSA | 5 | 7 | 12 |
| GSA | 12 | 11 | 23 |
| Pinpoint | 3 | – | 3 |
| total | 20 | 18 | 38 |

for example, we generated 1,000K random programs by Csmith at one round, and constructed the dynamic and static oracles based on these programs to test the analyzers. For those programs whose oracles were violated by the analyzers, we reduced these programs to report potential defects. This iterative testing process is useful for testing static analyzers. Because (1) we can know which program features are not well modeled according to the developers' feedback and thus should be disabled in the next round of testing to avoid reporting invalid defects, and (2) we need to wait for the developers to confirm and fix the reported defects, and thus we can avoid reporting and finding duplicated defects. This process explains why our experiment lasted for a long time.

In the whole testing process, we generated about 7,910K seed programs in total by Csmith. Among these seeds programs, 7,700K programs were used to generate dynamic oracles, and 210K programs were used to generate static oracles. In total, 13,500 programs with the dynamic oracles and 4,600 programs with the static oracles, respectively, reveal potential defects of the analyzers. We reduced these defect-triggering programs, filtered the programs with duplicated defects, manually inspected about 200 defects, and reported those unique defects.

### 4.2 Results of the Found Defects

Table 3 gives the results of found defects in CSA, GSA, and Pinpoint. Specifically, the dynamic oracle found 5, 12, and 3 defects in CSA, GSA, and Pinpoint, respectively, while the static oracle found 7 and 11 defects in CSA and GSA, respectively. Overall, the dynamic and static oracle found 20 and 18 defects, respectively. The number of defects found in CSA, GSA, and Pinpoint are 12, 23, and 3, respectively. In total, we found 38 defects in the three static analyzers.

Table 4 lists these found defects. Specifically, to facilitate understanding and discussion of these found defects, we classified them into different categories. To build the bug categories, we use open-card sorting approach [32]. To achieve this, two co-authors independently labeled the 38 defects with the categories which they felt accurately describes the root cause based on (1) the developers' feedback on these defects, (2) the fixing patch (if available) and (3) their own knowledge of these studied static analyzers. After the labeling, they discussed and cross-validated the labels. In this process, the authors followed the operational bug classification strategy [21, 36] to adjust the categories and achieve disjoint classification from low-level to high-level: a defect that is classified into the lower-level category indicating a narrower scope (*e.g.*, implementation errors) will not be considered in the higher-level one indicating a wider scope (*e.g.*, design choices). When the two co-authors could not reach consensus, the other co-authors participated in the discussion to help make the final decision. Finally, we built eight major categories from low-level to high-level issues, including "*Implementation Errors*", "*Defects in Heuristics*", "*Defects in Handling Loops*", "*Mishandling Language Features*", "*Overly Eager Assumption*", "*Defects Induced by Design Choices*", "*Defects Induced by Compiler Optimizations*", and "*Others*". In Table 4, column "Class" gives the classification of the defect. Column "Status" gives the status of the defect, including "*fixed*", "*confirmed*", "*duplicated*" (independently found by us but also reported by others), and "*pending*" (under active discussion between developers). Column "Oracle" indicates which type of oracle found this defect. Specifically, for the dynamic oracle type, the annotation of the "-" sign indicates that the corresponding defect is a false negative (FN), while the annotation of the "+" sign indicates that the corresponding

Table 4. Details of the 38 found defects. The classes include "Implementation Errors" (I), "Defects in Heuristics" (II), "Defects in Handling Loops" (III), "Mishandling Language Features" (IV), "Overly Eager Assumption" (V), "Defects Related to Design Choice" (VI), "Defects Related to Optimizations" (VII), and "Others" (VIII), respectively. The "-" sign indicates that the defect is a false negative, while "+" sign a false positive.

| # | Class | Status | Oracle | Analyzer | Defect Description |
|---|---|---|---|---|---|
| 1 | I | fixed | *Dynamic*$^{+}$ | GSA | missed the logic for spotting comparison of `&VAR` against `NULL`. |
| 2 | I | fixed | *Static* | GSA | missed the logic that `-X <= 0` is equivalent to `X >= 0`, where `X` is a integer. |
| 3 | I | confirmed | *Static* | GSA | does not treat the two pointers `&b` and `&b[(int)0]` as identical. |
| 4 | I | fixed | *Static* | CSA | does not treat `c >= b` and `b <= c` as the same in the true branch of `c >= b` where `b` and `c` are both pointers. |
| 5 | I | pending | *Static* | GSA | evaluates `e == d + 1` to be `UNKNOWN` with the fact that `e == &d`, `d` is an array and `e` is a pointer. |
| 6 | I | confirmed | *Dynamic*$^{+}$ | Pinpoint | mishandles modeling of intended two-dimensional array. |
| 7 | II | duplicated | *Static* | GSA | evaluate `a+3 > b+1` to be `UNKNOWN` but evaluate `a+2 > b+1` to be `TRUE` in the true branch of `if (a > b)`. |
| 8 | II | pending | *Static* | GSA | evaluates `a > b` to be `TRUE` but evaluates `b < a` to be `UNKNOWN` under the context that `a = c + 2` and `b = c + 1`. |
| 9 | II | confirmed | *Static* | CSA | evaluates the `b == 0` as `TRUE`, but evaluates `b+1 == 1` as `FALSE`, where `b` is a pointer. |
| 10 | II | confirmed | *Dynamic*$^{-}$ | CSA | inline defensive checks suppression heuristic. |
| 11 | III | confirmed | *Dynamic*$^{+}$ | CSA | limitation of modeling only 3 iterations of loops leads to CSA making assumptions, which lead to FP. |
| 12 | III | confirmed | *Dynamic*$^{+}$ | CSA | `widen-loop=true` option causes CSA to make a wrong assumption. |
| 13 | III | pending | *Dynamic*$^{-}$ | GSA | limitation in tracking variable reassignment in the `for` loop. |
| 14 | III | confirmed | *Dynamic*$^{-}$ | GSA | limitation in handling the initialization of an array with a `for` loop. |
| 15 | III | pending | *Dynamic*$^{+}$ | GSA | assumes it enters a `for` loop for twice which actually it can only enter it for once. |
| 16 | III | confirmed | *Dynamic*$^{-}$ | GSA | does not know the value of the iterator `d` after a vacant loop `for (d = -1; d !=0; ++d) {;}`. |
| 17 | III | pending | *Static* | GSA | evaluates `(c <= b) && (c != b) == false` to be `FALSE` with the fact `c >= b` within `goto` statement. |
| 18 | IV | confirmed | *Static* | GSA | gets confused about conditionals involving bit-fields. |
| 19 | IV | pending | *Static* | GSA | misses handling of `||` logic. |
| 20 | IV | confirmed | *Dynamic*$^{+}$ | Pinpoint | mishandles alias analysis because of global variables. |
| 21 | IV | confirmed | *Dynamic*$^{+}$ | Pinpoint | mishandles the two-dimensional array initialization in the form of `int a[2][7] = {9}`. |
| 22 | IV | fixed | *Dynamic*$^{-}$ | GSA | gives up exploring execution paths at the computed goto. |
| 23 | V | confirmed | *Static* | CSA | evaluates `if(-g.e && g.e)` to be `FALSE`, but it continues to do analysis of the code inside the if statement. |
| 24 | V | pending | *Static* | CSA | do not handle infinite recursion. |
| 25 | V | pending | *Static* | CSA | takes the true branch of `if(255UL == b)` with the fact that `b` is `int8_t` type. |
| 26 | VI | confirmed | *Dynamic*$^{+}$ | GSA | arguably improper design that every function could be top level function. |
| 27 | VI | confirmed | *Dynamic*$^{+}$ | CSA | improper assumption because of dead code. |
| 28 | VII | confirmed | *Dynamic*$^{-}$ | GSA | gives misleading messages at `-O2` && False Nagetive at `-O1` and `-O0`. |
| 29 | VII | confirmed | *Dynamic*$^{+}$ | GSA | `-O2` optimization leads to the false positive NPD warning in a loop. |
| 30 | VII | confirmed | *Dynamic*$^{-}$ | GSA | –Wanalyzer-out-of-bounds false negative with return arr[9]; at -O1 and above. |
| 31 | VIII | duplicated | *Static* | GSA | does not know `b > 0` under the if condition that `a>0 && b > a`. |
| 32 | VIII | pending | *Static* | CSA | does not know `b > 0` under the if condition that `a>0 && b > a`. |
| 33 | VIII | confirmed | *Dynamic*$^{+}$ | GSA | effected by deleting the unrelated code `int *d = 0;`. |
| 34 | VIII | duplicated | *Static* | GSA | div-by-zero false negative with `(d.b = 1) / f` and `(c = 1) % f` where `f` is zero. |
| 35 | VIII | confirmed | *Dynamic*$^{+}$ | CSA | does not support some C-library functions. |
| 36 | VIII | duplicated | *Dynamic*$^{+}$ | GSA | does not support some C-library functions. |
| 37 | VIII | confirmed | *Static* | GSA | evaluates `__analyzer_eval((a()<0)||(a()==0))` to be `TRUE`, but function `a()` is a unknown function. |
| 38 | VIII | pending | *Static* | CSA | assumes that `a()< 0||a()==0` to be `TRUE` in the true brach of `if(a()<= 0)` where `a()` is a external function. |

defect is a false positive (FP). In total, by using the dynamic oracle, we found 7 false negatives and 13 false positives. Column "Analyzer" denotes which static analyzer is affected. Column "Defect Description" gives a brief description of the defect.

Among all the 38 found defects, 28 have been confirmed by developers. Of these 28 confirmed defects, four were confirmed as duplicated (which were also reported by others) and four defects have been quickly fixed. In Section 4.3, we select some representative defects from each classified group to understand the limitations of the studied static analyzers.

### 4.3 Understanding the Found Defects

***Implementation Errors*** (Class I). Some defects are caused by implementation errors, such as failing to handle certain logic. We give two examples:

*(1) Mishandling the comparison between* `&var` *and* `NULL`. The defect #1 was found in GSA by the dynamic oracle. As illustrated in Listing 1, at line 5, `h[1][0]` is assigned by the variable `f` (*i.e.*, the

Listing 5. Program revealing defect #2.

```
1 void foo(int a) {
2   if (a >= 0) {
3     __analyzer_eval(a >= 0); // TRUE
4     __analyzer_eval(a-0 >= 0); // TRUE
5     __analyzer_eval(0-a <= 0); //
          UNKNOWN
6   }
7 }
```

Listing 6. Program revealing defect #7.

```
1 void foo(int a, int b) {
2   if (a > b) {
3     __analyzer_eval(a+0 > b+0); // TRUE
4     __analyzer_eval(a+1 > b+1); // TRUE
5     __analyzer_eval(a+2 > b+1); // TRUE
6     __analyzer_eval(a+3 > b+1); // UNKNOWN
7   }
8 }
```

memory address of e). Thus, the comparison expression at line 6 is equivalent to `NULL == &e` (note that g equals 0). However, GSA only handles the comparisons between `&e` and `NULL` when `NULL` is on the right-hand side (*i.e.*, `&e == NULL`). It mishandles when `NULL` is on *the left-hand side* of the equality operator (*i.e.*, `NULL == &e`), and thus assumes that `NULL == &e` could be true (see the error traces outputted by GSA in Listing 3). As a result, `f` is assumed as `NULL` and a false positive is warned at line 8. The developer fixed this defect by adding the logic to handle the case of left-hand side.

*(2) Miss handling comparisons against negated symbolic values.* This defect #2 was found in GSA through the static oracle. As Listing 5 shows, in the true branch of `if (a >= 0)`, `0 - a <= 0` is always true. However, GSA evaluates the expression at line 3 and line 4 both to be `TRUE`, but evaluates line 5 as `UNKNOWN`[5]. This means that GSA knows the facts that `a >= 0` and can infer `a - 0 >= 0` is true, but fails to infer that `0 - a <= 0` is also true. Evaluating `0 - a <= 0` to be `UNKNOWN` may result in FPs because static analyzers would assume `0 - a <= 0` could be `TRUE` or `FALSE`. The root cause of this defect is that the developer did not handle the comparisons against negated symbolic values. The developer fixed this defect by folding `0 - VAL` to `-VAL` and treating `-VAL <= 0` as equivalent to `VAL >= 0`. The defect #4 found in CSA is similar to this one. Specifically, CSA does not treat `c >= b` and `b <= c` as equivalent in the true branch of `c >= b` (b and c are the variables of pointers). To fix this defect, CSA's developers added a patch to the constraint manager by looking up the commutative forms of the expressions like `c >= b` and treats these expressions as equivalent.

Defect #1~#6 are all due to some implementation errors. Of these, defect #1, #3, #4, and #5 are all related to legal pointer or array operations. Defect #2 is related to arithmetic operations. Pinpoint also has implementation errors. In defect #6, Pinpoint intended to model the first two elements of each dimension of the two-dimensional array, but dropped the information of the second element when encountering the complex loop logic. These implementation errors in these static analyzers lead to incorrect evaluation of boolean expressions, which could lead to false positives or negatives.

***Defects in Heuristics*** (Class II). The developers may adopt some heuristics in handling specific cases in static analysis which however may lead to unexpected effects. Here are the two examples:

*(1) GSA's heuristics for handling* `a+N > b+M`. Defect #7 (see Listing 6) is found by the static oracle. Under the condition `a > b` (line 2), GSA infers that `a+0 > b+0`, `a+1 > b+1` and `a+2 > b+1` to be `TRUE` (lines 3, 4, and 5), but evaluates `a+3 > b+1` to be `UNKNOWN`. These evaluation results are inconsistent. The root cause is that GSA's *constraint manager* plays fast-and-loose in places. The GSA's developer confess that the heuristics used to infer `a+N > b+M` when `a > b` and `N >= M` is imprecise. Because in strict sense `a+1 > b+1` is not always true. For example, if `a == INT_MAX` and `b == INT_MAX-1`, `a > b` is TRUE, but `a+1` becomes an undefined behavior (`a+1` should be treated as `INT_MIN`). In this case, `a+1 > b+1` becomes `INT_MIN > INT_MAX`, which should be evaluated to be `FALSE`. Due to this defect, the developer is going to hand this constraint tracking off to an SMT solver.

[5]We catch the FALSE evaluation in the original program. After reducing, we find that the root cause is that GSA evaluates `0 - a <= 0` as `UNKNOWN`, leading to some complex expressions are evaluated to be `FALSE` in the original program.

Listing 7. Program revealing defect #9.

```
1 void foo() {
2   int *b = (void *)0;
3   if (b == (void *)0) {
4     clang_analyzer_eval(b == (void *)0); // TRUE
5     clang_analyzer_eval(b+1 == (void *)1 ); // FALSE
6     int *p = (void *)0;
7     if (b + 1 != (void *)1)
8       *p = 100;
9 }}
```

Listing 8. Program revealing defect #11.

```
1 void foo(int *e) {
2   *e = (0 == e); // warning of NPD
3 }
4
5 void main(){
6   int d[4];
7   for (int c = 0; c < 4; c++){}
8     foo(d);
9 }
```

*(2) CSA's heuristics for assisting null pointer dereference.* Defect #9 (see Listing 7) was found by the static oracle. CSA evaluates the condition `b == (void*)0` on line 4 as `TRUE`, but evaluates `b+1 == (void *)1` on line 5 as `FALSE`. However, in reality `b+1 == (void *)1` should be `TRUE`. Due this incorrect evaluation, CSA concludes that line 8 is reachable and reports a false positive of NPD on line 8. The CSA's developer confess that this an "*ugly hack*" that was implemented to assist the NPD analysis. The developer implemented any arithmetic operation on a concrete null pointer as always resulting in a literal concrete null pointer. Due to this heuristic, `b+1` is treated as an NPD because `b` is a null pointer and the data behind `b` is accessed. The developer responded that "*Some effort is required to undo this heuristic while preserving the null dereference checker behavior.*" We find that Pinpoint also implements the similar heuristic, while GSA does not have this issue.

In summary, some heuristics in the constraint manger may lead to imprecise analysis results when handling numerical comparisons (see defect #7 and #8). Some heuristics in handling pointer arithmetic operations may also lead to imprecision (see defect #9). In addition, defect #10 shows that some heuristics intending to suppress false positives may unexpectedly bring false negatives.

***Defects in Handling Loops*** (Class III). Handling loops is difficult for static analysis. The static analyzers face the problems of determining the iterations of loops, the complex logic within loops, and the dependency on the input data. The analyzers usually use some strategies to mitigate the challenges, but these strategies may bring some unexpected effects. Here are the three examples:

*(1) Defects in modeling the iterations of loops.* Defect #11 (see Listing 8) was found by the dynamic oracle. CSA reports a warning of NPD at line 2 by assuming `e` is `null`. However, since function `foo` is called by `main` with the argument `d`, `e` cannot be `null`. Thus, this warning is spurious. On the other hand, we find that CSA will not give this warning if the loop condition (at line 7) is set to `c < 3`. This inconsistency may confuse the CSA's users. The developers explained the root cause of the defect: CSA by default models at most three iterations of loops before stopping on that path. In Listing 8, since the number of loop iterations at line 7 exceeds three, the function call `foo(d)` will not be analyzed. In this case, function `foo` will be treated as a top-level function. When `foo` is analyzed as a top-level function, CSA assumes that `e` can be given any value (including `null`).

*(2) Defect in widening loops.* Defect #12 (see Listing 9) was found by the dynamic oracle. When the "`widen-loops=true`" option is enabled, CSA reports a false positive warning of NPD at line 10. We can see that variable `f` refers to the address of variable `e` at line 7, and thus `f` cannot be `null` at line 10 (`f` is never overwritten). The defect is caused by some bug in widening loops (this false positive will not appear without enabling the "`widen-loops=true`" option). During widening loops, CSA sees a reference to variable `f` (line 10) which is part of the expression `*f = a(f==0)` within the innermost loop. However, CSA does not consider that the expression `f==0` never overwrites the value of `f`, so CSA assumes `f` may get overwritten by a fresh symbol. Then, when CSA reaches the

Listing 9. Program revealing defect #12.

```
1  int *a(int i) {
2    int *n = (void *)0;
3    return n;
4  }
5  void main() {
6    int d, *e;
7    int **f = &e;
8    for (int g = 0; g < 3; g++) {
9      for (d = 2; d; d--) {
10       *f = a(f == 0); // warning of NPD
11 }}}
```

Listing 10. Program revealing defect #13.

```
1  void main() {
2    int e = 1;   int *f;
3    for (int i = 0; i < 1; i++) {
4      e = 0;
5      __analyzer_eval(0 == e); // TRUE
6    }
7    __analyzer_eval(0 == e); // UNKNOWN
8    f = e;
9    __analyzer_eval(0 == f); // UNKNOWN
10   *f = 1; // miss reporting NPD
11 }
```

expression `f==0`, it no longer knows that `f` refers to `&e`, and `f` is not `null`; so CSA splits the path, and it ends up on a path where `f` is assumed to be `null`.

*(3) Defect in loosely handling of loops.* Defect #13 (see Listing 10) is a false negative of GSA found by the dynamic oracle (we give some debug checks in Listing 10 to facilitate understanding this defect). An NPD error exists at line 10, but GSA misses this NPD. GSA evaluates `0 == e` (line 5) to be `TRUE`, but evaluates `0 == e` (line 7) and `0 = f` (line 9) to be `UNKNOWN`. This case shows that GSA lacks the ability to track the reassignment of variable `e` within the loop. GSA symbolically executes the loop body once, so it should know the value of `e` in the loop body. However GSA does not retain the symbolic state after the loop, so it loses the value of `e` after the loop body. Even if the optimization option `-O2` is enabled, GSA still cannot find this NPD because the optimization passes do not optimize the loop out. In contrast, CSA can find this NPD because it fully unrolls such simple loops that satisfy the attributive condition [35].

In summary, the static analyzers use different strategies to mitigate the challenges of analyzing loops. For instance, CSA sets the default number of iterations for loops as three (revealed by defect #11). CSA also adopts some strategies to widen loops [35], but these strategies have some faults (revealed by defect #12). Pinpoint by default unrolls loops for at most two iterations (revealed by defect #6). GSA handles loops in a different yet simpler manner. It performs the symbolic execution on the loop body once, but does not retain the symbolic state after the execution (revealed by defect #13). Additionally, since GSA is implemented as an inter-procedural optimization (IPA) pass, and some code optimizations will be performed before static analysis. With -O2 optimization, defect #14, #15, and #16 will not appear, while defect #13 always exists on different levels of optimizations.

***Mishandling Language Features*** (Class IV). Mishandling some language features may lead to false negatives or positives of static analyzers. Here are the three examples:

*(1) Mishandling of bit-fields.* Defect #18 (see Listing 11) was found in GSA by the static oracle. The variable `b.d` is assigned as zero at line 5. GSA evaluates `b.d` (line 8) to be `FALSE`, which is correct. But GSA evaluates `b.d == 0` (line 9) and `b.d != 0` (line 10) to be `UNKNOWN`, which is imprecise. Moreover, if we use `__analyzer_dump_path()` to compute the reachability of the true branch of `if(b.d)`, GSA says it is reachable. This conclusion is contradictory with the fact that `b.d` is zero. This phenomenon indicates that GSA mishandles bit-fields. This mishandling could lead to a false positive warning of NPD at line 12. In fact, GSA only handles specific widths of bit-field. Specifically, if the bit-field at line 2 is modified to 1, 8, 16, or 32, GSA can give correct conclusion that the true branch of `if(b.d)` is not reachable. However, it cannot handle other widths of bit-field.

*(2) Mishandling of the logical OR (`||`).* Defect #19 (see Listing 12) was found in GSA by the static oracle. GSA assumes line 12 is reachable, and thus gives a warning of NPD against variable `d` at line 12. However, this warning is a false positive. Because taking the true branches of `if(!a)` (line 9) and

Listing 11. Program revealing defect #18.

```
1  struct a{
2    int d: 10;
3  } b;
4  void foo(){
5    b.d = 0;
6    int * p;
7    if (b.d){
8      __analyzer_eval(b.d); // FALSE
9      __analyzer_eval(b.d==0); // UNKNOWN
10     __analyzer_eval(b.d!=0); // UNKNOWN
11     __analyzer_dump_path(); // Reachable
12     *p = 42; // warning of NPD
13   }
14 }
```

Listing 12. Program revealing defect #19.

```
1  int foo(bool a, bool b) {
2    int *c = 0;
3    int *d = 0;
4    if (a || b){
5      __analyzer_eval(a); // UNKNOWN
6      __analyzer_eval(b); // UNKNOWN
7      __analyzer_eval(a||b); // UNKNOWN
8
9      if (!a){
10       if (!b){
11         __analyzer_eval(a||b); // UNKNOWN
12         *d = 0; // warning of NPD
13       }
14 }}}
```

`if(!b)` (line 10) is contradictory with the GSA's assumption that `a || b` (line 4) could be true. In addition, GSA correctly evaluates the expressions at lines 5 and 6 to be `UNKNOWN`, but it imprecisely evaluates the expressions at lines 7 and 11 to be `UNKNOWN`. This phenomenon means GSA's constrain manager does not properly handle the logical OR. In contrast, both CSA and Pinpoint correctly conclude that line 12 is dead code and do not report the NPD warning.

*(3) Mishandling of computed* `gotos`. Defect #22 (see Listing 13) was found in GSA by the dynamic oracle. At line 2, array `arr` is initialized with the addresses of two goto labels `x` and `y`. We can see that label `x` is reachable, and there is an NPD error at line 7. However, GSA does not report this NPD because GSA gives up exploring execution paths at the computed `gotos`. GSA ignores the CFG edges with the flag `EDGE_ABNORMAL`. This bug has been fixed by the GSA's developer.

In summary, the static analyzers may mishandle some language features, *e.g.*, bit-fields (defect #18), logical OR (defect #19), global variables (defect #20), the array initialization by brace-enclosed lists (defect #21) and computed `gotos` (defect #22). Among these features, global variables are one of the main reasons of inducing false positives, because the analyzers usually assume that global variables could be any value at any place when used.

***Overly Eager Assumption*** (Class V). In order to guarantee the soundness, the analyzers usually make assumptions when they cannot find all the necessary information. However, we find that the analyzers may still make assumptions when sufficient information is available (revealed by defect #23, #24, and #25). This kind of overly eager assumption can lead to false positives.

For example, CSA sometimes assume both branches of a conditional statement are reachable. Here are the two examples. Defect #23 (see Listing 14) was found in CSA by the static oracle. This defect leads to a false positive warning of NPD at line 9. CSA can correctly evaluate the expression `-g.e && g.e` at line 7 to be `FALSE`, but it simply assumes the true branch of the conditional statement `if (-g.e && g.e)` (line 8) to be reachable and reports the NPD warning. This is a typical overly eager assumption in CSA. We tried this case with GSA and Pinpoint. Both GSA and Pinpoint do not make such eager assumptions on this case.

Defect #25 (see Listing 15) was also found in CSA by the static oracle. Since the value range of type `int8_t` is from -128 to 127, only the false branch of `if(255UL == b)` (line 3) is reachable. However, CSA assumes that both branches are reachable even if CSA knows that the expression `255UL == b` (either at line 4 or line 6) is evaluated to be `FALSE`.

***Defects Related to Design Choices*** (Class VI). The design choices made by the static analyzers' developers may induce false positives or negatives. For example, GSA treats every function as

Listing 13. Program revealing defect #22.

```
1  void foo(int b) {
2    int *arr[2] = {&&x, &&y};
3    goto *arr[b];
4
5  x:
6    arr[0] = (void *)0;
7    *arr[0] = 42; // miss reporting NPD
8    return;
9  y:
10   return;
11 }
12 int main() { foo(0); }
```

Listing 14. Program revealing defect #23.

```
1  union d {
2    int e;
3  };
4
5  int main() {
6    union d g = {};
7    clang_analyzer_eval(-g.e && g.e); // FALSE
8    if (-g.e && g.e){
9      *(int *)0; // warning of NPD
10   }
11 }
```

Listing 15. Program revealing defect #25.

```
1  void foo(int8_t b) {
2
3    if (255UL == b)
4      clang_analyzer_eval(255UL==b); // FALSE
5    else
6      clang_analyzer_eval(255UL==b); // FALSE
7  }
```

Listing 16. Program revealing defect #26.

```
1  int a = 1; int *b = &a;
2  void c() {  int f; f = *b; } // warning of NPD
3  void e() {
4    if (0 == b)
5      int *g = (void *)0;
6  }
7  void main() { e(); c(); }
```

an entry point for static analysis. This design choice may lead to false positives. Defect #26 (see Listing 16) was found in GSA by the dynamic oracle. We can see that function `e` and `c` are called by function `main`. However, GSA warns an NPD (corresponding to `f = *b`) at line 2. Because GSA treats function `c` might be called externally, and thus does not consider the constraint `b == &a` (line 1). In this way, the global variable `b` could be `NULL` (at line 2). In contrast, CSA conducts a topology analysis between function calls. CSA treats `main` as the entry point of static analysis and other functions called by `main` would not be treated as top-level functions. Pinpoint also does a similar topology analysis but analyzes the functions in a bottom-up matter. Both CSA and Pinpoint do not give the false positive warning on this case.

***Defects Related to Optimizations*** (Class VII). Defect #30 (see Listing 17) shows that the optimizer may affect the static analysis results of GSA. This defect was found by applying the dynamic oracle to the out-of-bound (OOB) checker. The code at line 5 in Listing 17 contains an OOB. With the `-O0` optimization option, GSA can successfully report this OOB. However, with the `-O1` or `-O2` option, GSA misses this OOB. The developer confirmed this defect and pointed out that "this is a side-effect of how late the analyzer runs after optimizations". By inspecting the GIMPLE IR, we find that the expression `arr[9]` has been optimized out before GSA sees it when the `-O1` or `-O2` option is given. Due to this defect reported by us, the GSA's developers opened a dedicated thread to discuss the topic "*should the analyzer run earlier?*". Since the optimizer may remove the code containing undefined behavior before GSA ever sees it, various program bugs are showing up where GSA fails to warn on clearly wrong code. Thus, there is a tension between the warnings and optimization: the optimizer takes advantage of undefined behavior (assuming it's not present), while the static analyzer should complain about the presence of undefined behavior. On the other hand, it is better to report warnings to the users in a form closer to that in which they wrote the code. Therefore, it is preferable to run GSA at an earlier stage.

Listing 17. Program revealing defect #30.

```
1 static int f();
2 int f() {
3   int arr[1];
4   for (int h = 0; h < 1; h++) arr[h] = 3;
5   return arr[9]; // miss reporting OOB
6 }
7 int main() { f(); }
```

Listing 18. Program revealing defect #33.

```
1 void foo(int* e) {
2   if(e == 0){
3     int *d = 0; // the extraneous code
4     *e = 1; // an NPD warning
5   }
6 }
```

***Others*** (Class VIII). There are other types of defects that cannot be classified into the categories discussed before. We briefly illustrate these defects. Defect #31 and #32 are respectively caused by the limitations of the constraint managers in GSA and CSA. The managers are usually designed as lightweight solvers and are difficult to solve complex constraints. Defect #35 and #36 were found in CSA and GSA since they fail to model C-library functions like `printf`. Because function `printf` is a variadic function which is not easy to model. In defect #33 (see Listing 18), GSA warns an NPD at line 4 assuming variable `e` could be `null`. However, if we remove the extraneous code `int *d = 0` at line 3, the NPD warning disappears, which is unexpected.

## 5 DISCUSSION AND LESSONS LEARNED

This section discusses our additional experience of finding defects in the three studied static analyzers and summarizes some lessons learned.

***Static analyzers should clarify the nature and extent of unsoundness.*** Virtually all modern static analyzers, including CSA, GSA and PinPoint which we studied, are unsound in practice [24]. It means, on the one hand, these static analyzers over-approximate most common language features (*i.e.*, modeling all the possible program behaviors) to ensure soundness. On the other hand, some specific language features, well known to experts in the area, are deliberately under-approximated to improve precision and/or scalability — such engineering compromises undermine the soundness. For example, defect #9 in our study is caused by the under-approximation of pointer arithmetic in CSA; defect #18 is induced because GSA only models some specific width of bit-field. Thus, static analyzers' developers (researchers) should clarify the nature and extent of unsoundness so that the wider community of users who expects to use static analyzers as black-box can better understand the tool limitations [24]. Our work also contributes in this regard to help the users gain a better understanding of these static analyzers (and static analysis).

***Compiler optimizations affect GSA's static analysis results.*** GSA is implemented as an inter-procedural analysis (IPA) pass like many compiler optimizations in GCC. Its static analysis results could be affected by the optimizations preceding GSA. Because different optimization levels (*e.g.*, `-O0`, `-O1` and `-O2`) will enable different optimization passes before GSA is invoked. On the positive side, GSA can leverage the optimization passes, such as dead code elimination and constant propagation, to simplify the code before doing static analysis. For example, in our study, when `-O2` is enabled, GSA can successfully avoid false negatives (defects #14, #16) and positives (defect #15) because some code in the loops are simplified. On the negative side, the optimization passes may introduce false negatives by optimizing out important code snippet. For example, defect #30 (which induces a false negative) only appears when `-O1` and `-O2` are enabled. One GSA developer commented in the bug report of defect #30 that "*that's a strong argument that the analyzer should run earlier.*" As a side note, CSA and Pinpoint do the static analysis on the unoptimized LLVM bitcode.

***Constraint solving in static analyzers.*** GSA and CSA use internal (lightweight) constraint solvers to analyze path feasibility in their default settings. In the case of complex path constraints, the

solvers may give up and thus induce spurious warnings (*i.e.*, false positives). To counter this issue, CSA can be configured with Z3 [11], a potent constraint solver, to refute spurious warnings [28]. To examine whether the 12 defects found in CSA (see Table 3) could be refuted by Z3, we ran CSA with Z3 on the 12 corresponding defect-triggering programs. We find that only defect #25 can be eliminated. It indicates that the remaining 11 defects are non-trivial because they are only relevant to the core analysis module rather than the internal solver. As a side note, GSA is incorporating Z3 to address the defects similar to defect #7 reported by us, while Pinpoint uses Z3 by default.

***Implementation errors should be avoided.*** All the three studied analyzers are affected by implementation errors. We discuss some cases of the category *Implementation Errors* in detail in Section 4.3. Upon examining these implementation errors, we observe that developers may overlook certain intricate analysis logic, thereby rendering static analyzers incapable of analyzing code that encompasses such complexities. However, in such cases, static analyzers will designate the logically interrelated expression as `UNKNOWN` to ensure the soundness, thus leading to false positives.

***Cross-checking the three analyzers on the defect-triggering programs.*** We used the 38 defect-triggering programs to compare the capabilities of three static analyzers. Because of Pinpoint's lack of debugging checkers, for those static oracles, we insert NPDs into those defect-triggering programs to find whether Pinpoint shows up FPs or FNs. The results are given in *Table 1 in the supplementary material* [46]. Apart from its own three defects, Pinpoint also fails on defect #9, #14, #16, and #24. Overall, Pinpoint produces fewer numbers of false positives. Interestingly, the programs revealing defect #14 and #16 in GSA trigger different defects in Pinpoint. In addition to its own defects, CSA is also affected by defect #7, #31, and #36. These three defects are included in its own defects. Additionally, CSA is also affected by defect #20, which is related to global variables. Similarly, GSA is also affected by defect #32 and #35, which are also included in its own defects.

***The dynamic and static oracles are complementary in finding defects.*** The two types of oracles used in our work are complementary in finding defects in static analyzers. For example, the dynamic oracle can never find the defect like defect #19, because it requires a dynamic execution path reaching line 12. But line 12 (in Listing 12) is not reachable. On the other hand, the fault detection ability of the static oracle is limited to generating equivalent boolean expressions from selected conditional statements. The static oracle is difficult to find the defects of handling the `for` loops (*e.g.*, defect #11). Because these analyzers face challenges in modeling the iterations of loops.

***Testing coverage of the automated oracles.*** The 38 defects found by our work have demonstrated the effectiveness of the two automated oracles. To investigate the testing coverage of these two automated oracles, we measured (1) the achieved code coverage of the studied static analyzers, and (2) the number of checkers which can be tested by the dynamic oracle.

We used Gcov [41] to measure the line coverage achieved by the randomly generated programs with the dynamic and static oracles based on the setup in Section 4.1. We measured the coverage of core analysis modules in CSA (`tools/clang/lib/StaticAnalyzer/Core/`) and GSA (`gcc/analyzer/`). The dynamic oracle achieved 49.6% and 44.6% of the line coverage of CSA and GSA respectively. The static oracle achieved 45.9% and 43.5% of the line coverage of CSA and GSA respectively.

The dynamic oracle is not limited to the NPD checker used in Section 4.1. In theory, the dynamic oracle can leverage any checker whose targeted bugs can be caught at runtime or transformed into runtime errors. We find that 40 out of all the 48 checkers (83.3%) in GSA and 14 out of all the 14 core checkers (100%) in CSA can be tested by the dynamic oracle. It shows that most checkers in CSA and GSA can be tested by the dynamic oracle. To demonstrate the feasibility, we used the out-of-bound (OOB) checkers of CSA and GSA for additional experiments. The OOB checkers found some similar defects which were found by the NPD checkers. But the OOB checkers also found one unique defect #30 in GSA, which cannot be found by the NPD checkers.

Table 5. Key Differences between the prior work and ours in testing static analyses.

| Work | Which static analyses were tested? | What kinds of test oracles [2] are used? |
|---|---|---|
| Cuoq *et al.* [10] | value analysis, constant propagation, slicing | Pseudo-oracles |
| Wu *et al.* [47] | alias analysis | Pseudo-oracles |
| Taneja *et al.* [37] | data-flow analysis | Pseudo-oracles |
| Midtgaard *et al.* [27] | type analysis | Specification |
| Bugariu *et al.* [3] | abstract domain | Specification |
| Our work | off-the-shelf static analyzers | Pseudo-oracles & Metamorphic Relations |

***Both False negatives and positives are important.*** Both false negatives and positives are important to be tackled in building reliable static analzyers. On one hand, it is important to mitigate the false negatives to avoid missing true program bugs. On the other hand, the issue of false positives is one of the main reasons that engineers ignore the tool's warnings, which is the lesson Caitlin et al. [30] learned from building static analysis tools at Google. In our study, we benefit from both false positives and false negatives to have a deep understanding of static analyzers. For example, the false positive of defect #9 is caused by the under-approximation of pointer arithmetic in CSA. The false negative of defect #30 is caused by the optimization passes in GSA.

## 6 RELATED WORK

In the literature, a number of work has been conducted in finding soundness and precision bugs of static analyses. Table 5 illustrates the key differences between the prior work with ours in terms of (1) *which static analyses were tested* and (2) *what kinds of test oracles [2][6] are used.* Cuoq *et al.* [10] use Csmith to generate random programs to test some specific static analyses in Frama-C [40][7], *i.e.*, *value analysis* (based on abstract interpretation), *constant propagation* (a program transformation plugin based on value analysis), and the *slicing* plugin. Specifically, Cuoq *et al.* compare the values inferred by static analyses with those from actual program executions as test oracles to find bugs. Since Cuoq *et al.* test low-level static analyses rather than off-the-shelf static analyzers like CSA, GSA and Pinpoint. Their oracles cannot be applied in our setting. Moreover, the bugs found by Cuoq *et al.* are different from those found by us. For example, Cuoq *et al.* find the bugs in *or* related to the front-end, alarm emission, program pretty-printing and slicing. In contrast, the bugs found by us are much more diverse. Cuoq *et al.* do not discuss the found bugs in detail, while we carefully classify and illustrate the found bugs.

There are work testing other specific static analyses, *e.g.*, alias analysis [47], data-flow analysis [37], type analysis [27] and abstract domains [3, 6, 26]. Wu *et al.* [47] compare the pointer addresses inferred by LLVM's *alias analyses* with those dynamically tracked by running programs as test oracles to find errors. Taneja *et al.* [37] compute sound and maximally precise static analysis results by using an SMT solver, and cross-check against the analysis results of LLVM's *data-flow analyses* to find bugs. Midtgaard *et al.* [27] use several algebraic lattice properties as test oracles to test a *type analysis* for the Lua programming language. Bugariu *et al.* [3] use several mathematical properties of abstract domains as test oracles to validate the soundness and precision of *numerical abstract domains*, the core of all static analyzers based on abstract interpretation.

Our work has two key differences with these preceding prior work in Table 5. First, our work tests off-the-shelf static analyzers for finding bugs rather than specific static analyses. Second, our work proposes two types of novel automated oracles. At the high-level, the dynamic oracle compares the static analysis results with the results obtained by dynamically running programs

[6]We use the terms of different types of test oracles defined in [2] to name the test oracles used by these work in Table 5. We will discuss the concrete test oracles used by these work in Table 5 in detail.

[7]Note that Frama-C is an analysis framework for C language including multiple static analyses. It is not an off-the-shelf static analyzer for finding bugs like CSA, GSA, and Pinpoint.

like [1, 6, 10, 47]. But the concrete forms of test oracles are different. The static oracle is constructed based on metamorphic relations, which have not been explored by these prior work.

In the literature, Wang *et al.* [45] and Zhang *et al.* [50] test some off-the-shelf static code analyzers like PMD, SpotBugs and SonarQube. However, these analyzers are only *linters* [25] which mainly adopt syntax-based pattern matching to find program flaws like best practice violations and coding issues, while the analyzers studied in our work adopt flow-based analysis techniques to find deep program bugs. Therefore, all the bugs found in these linters are caused by inadequate syntactic pattern matching (see Section 3.2/3.3 in [45] and Section 5.2 in [50]), which are different from ours. Finding the defects in these static analyzers can help improve the soundness and precision. A recent survey summarizes different techniques to mitigate spurious static analysis warnings [14].

There are also some work on testing compilers [12, 19] and program analyzers (*e.g.*, software model checkers [17, 49], symbolic execution engines [16]). Vu Le *et al.* [19] propose an approach for validating optimizing compilers by exploiting the close interplay between dynamically executing a program on some test inputs and statically compiling the program to work on all possible inputs. Even-Mendoza *et al.* [12] apply coverage-guided gray-box fuzzing to find crashing bugs in compilers and program analyzers. Zhang *et al.* [49] propose a fully-automated branch reachability fuzzing approach to test the soundness of software model checkers. Klinger *et al.* [15] use differential testing to find soundness and precision bugs in software model checkers. Kapus *et al.* [16] find bugs in symbolic execution engines by crosschecking the symbolic execution runs against native runs.

## 7 CONCLUSION

We introduce two novel types of automated oracles by leveraging the dynamic program execution and inferred static analysis results to help find the defects in static analyzers. We applied the two oracles on three state-of-the-art static analyzers which adopt different static analysis techniques. We found 38 unique defects in these analyzers, 28 of which have been confirmed and 4 have been fixed. The results demonstrate the effectiveness of the two oracles. We conducted a case study to illustrate these found defects followed by several lessons learned for improving and better understanding these static analyzers.

## ACKNOWLEDGMENTS

We thank the anonymous FSE reviewers for their valuable feedback. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002), CCF-AFSG Research Fund, and "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100.

## REFERENCES

[1] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3088515.3088521

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering (TSE)* (2014), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[3] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 768–778. https://doi.org/10.1145/3238147.3240464

[4] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the Program Analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 765–768. https://doi.org/10.1145/2889160.2889206

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[6] Ignacio Casso, José F Morales, Pedro López-García, and Manuel V Hermenegildo. 2020. Testing your (static analysis) truths. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 271–292. https:

//doi.org/10.1007/978-3-030-68446-4_14

[7] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27. https://doi.org/10.1145/3143561

[8] Xiao Cheng, Jiawei Wang, and Yulei Sui. 2024. Precise Sparse Abstract Execution via Cross-Domain Interaction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 109, 12 pages. https://doi.org/10.1145/3597503.3639220

[9] Clang Static Analyzer. 2023. https://clang-analyzer.llvm.org/, Last accessed on 2023-04-10.

[10] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods*. 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. https://doi.org/10.5555/1792734.1792766

[12] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3597926.3598130

[13] GCC Static Analyzer. 2023. https://gcc.gnu.org/wiki/StaticAnalyzer, Last accessed on 2023-04-10.

[14] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Trans. Software Eng.* 49, 12 (2023), 5154–5188. https://doi.org/10.1109/TSE.2023.3329667

[15] Gábor Horváth, Réka Nikolett Kovács, and Péter Szécsi. 2022. Report on the differential testing of static analyzers. *Acta Cybernetica* 25, 4 (2022), 781–795. https://doi.org/10.14232/actacyb.282831

[16] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 590–600. https://doi.org/10.1109/ASE.2017.8115669

[17] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/3293882.3330553

[18] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*. 1–20.

[19] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 216–226. https://doi.org/10.1145/2594291.2594334

[20] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[21] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*. 25–33. https://doi.org/10.1145/1181309.1181314

[22] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 544–555. https://doi.org/10.1145/3533767.3534380

[23] Jiangchao Liu, Jierui Liu, Peng Di, Diyu Wu, Hengjie Zheng, Alex Liu, and Jingling Xue. 2023. Hybrid Inlining: A Compositional and Context Sensitive Static Analysis Framework. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 114–126. https://doi.org/10.48550/arXiv.2210.14436

[24] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[25] Panagiotis Louridas. 2006. Static Code Analysis. *IEEE Softw.* 23, 4 (2006), 58–61. https://doi.org/10.1109/MS.2006.114

[26] Magnus Madsen and Ondřej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 38–48. https://doi.org/10.1145/3213846.3213847

[27] Jan Midtgaard and Anders Møller. 2015. QuickChecking Static Analysis Properties. In *8th IEEE International Conference on Software Testing Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 1–10. https://doi.org/10.

1109/ICST.2015.7102603

[28] Mikhail R. Gadelha, Enrico Steffinlongo, Lucas C. Cordeiro, Bernd Fischer, and Denis Nicole. 2019. SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyzer. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 11–14. https://doi.org/10.1109/ICSE-Companion.2019.00026

[29] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Conference on Programming Language Design and Implementation (PLDI)*. 335–346. https://doi.org/10.1145/2254064.2254104

[30] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[31] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 693–706. https://doi.org/10.1145/3296979.3192418

[32] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[33] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse Flow-Sensitive Pointer Analysis for Multithreaded Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*. 160–170. https://doi.org/10.1145/2854038.2854043

[34] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266. https://doi.org/10.1145/2892208.2892235

[35] Péter György Szécsi, Gábor Horváth, and Zoltán Porkoláb. 2022. Improved Loop Execution Modeling in the Clang Static Analyzer. *Acta Cybernetica* 25, 4 (2022), 909–921. https://doi.org/10.14232/actacyb.283176

[36] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705. https://doi.org/10.1007/s10664-013-9258-8

[37] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 81–93. https://doi.org/10.1145/3368826.3377927

[38] The Clang Team. 2024. Available Checkers for the Clang Static Analyzer. https://clang.llvm.org/docs/analyzer/checkers.html. Accessed: 2024-02-20.

[39] The Clang Team. 2024. Debug Checkers for the Clang Static Analyzer. https://clang.llvm.org/docs/analyzer/developer-docs/DebugChecks.html. Accessed: 2024-02-20.

[40] The Frama-C Team. 2024. Frama-C home page. https://frama-c.com/. Accessed: 2024-02-20.

[41] The GCC Team. 2024. Gcov — a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Accessed: 2024-02-20.

[42] The GCC Team. 2024. Options That Control GCC Static Analysis. https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html. Accessed: 2024-02-20.

[43] The GCC Team. 2024. Special Functions for Debugging the GCC Static Analyzer. https://gcc.gnu.org/onlinedocs/gccint/Debugging-the-Analyzer.html. Accessed: 2024-02-20.

[44] The Clang Team. 2019. Clang 9 documentation: LibTooling, Retrieved Feb. 2019 from https://clang.llvm.org/docs/LibTooling.html.

[45] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find bugs in static bug finders. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 516–527. https://doi.org/10.1145/3377811.3380380

[46] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, Yulei Sui. 2024. Supplementary Material. Retrieved 2024-05-03 from https://github.com/Geoffrey1014/SA_Bugs/blob/master/figures/Supplementary-Materials.pdf.

[47] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective dynamic detection of alias analysis errors. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. 279–289. https://doi.org/10.1145/2491411.2491439

[48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294. https://doi.org/10.1145/1993498.1993532

[49] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 763–773. https://doi.org/10.1145/3338906.3338932

[50] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 237–249. https://doi.org/10.1145/3611643.3616272

[51] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, and Alex X. Liu. 2022. Field-Based Static Taint Analysis for Industrial Microservices. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 149–150. https://doi.org/10.1145/3510457.3513075

[52] Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, Alex X. Liu, and John C. S. Lui. 2023. Scalable Compositional Static Taint Analysis for Sensitive Data Tracing on Industrial Micro-Services. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 110–121. https://doi.org/10.1109/ICSE-SEIP58684.2023.00015

Received 2023-09-28; accepted 2024-04-16