Polystat vs. Others: Static Analysis Tools Benchmark Report

Polystat Team

June 19, 2022

Abstract

This document presents a preliminary report on the performance of Polystat compared with other static analysis tool across several types of defects in object-oriented programs.

1 Introduction

Polystat is a static analyzer for object-oriented languages, based on φ-calculus. The calculus and EO programming language have been introduced by Bugayenko (2021a) in an attempt to distill his vision of the essence of object-oriented programming. EO is a very minimalistic programming language and is argued to be capable of representing faithfully many object-oriented structures commonly occurring in modern software engineering.

At the time of writing, Polystat consists of two main parts: a) `polystat` itself offering the primary entry point and the module for division-by-zero defect detection, and b) `odin` module for object-related defect detection. Polystat is capable of analyzing only EO programs.

For comparison with Polystat, we use popular open-source analyzers that are capable to detect defects in C++ programs: Clang-Tidy, Cppcheck, and SVF. In this document, we compare analyzers using several static analysis metrics and interpret the benchmark report.

As a result of the research we have established that: a) among the analyzers participating in the comparison, only Polystat can find OOP-specific defects, b) Polystat shows 100% recall for these OOP-specific defects, and c) Polystat shows comparable performance for the common type of defects. Table 1 demonstrates the numbers collected.

2 Materials and Methods

In this section we describe our approach to benchmarking static analysis tools for C++ and EO. We first describe the general methodology. Then, we present a method of translating programs from C++ to EO. Finally, we give descriptions of three types of defects that are of special interest in this report.

2.1 Comparing static analysis tools

For this report, we use a direct and simple approach of comparing static analysis tools. Put simply, we have a collection of example programs marked as “good” (meaning that the program is defect-free) or “bad” (meaning that it has some defect). We run static analysis tools on these programs and check whether the tool agrees with the markings.

The approach has several limitations, such as ignoring actual type, location, and confidence level of defect reported by the tool, as well as supporting programs with multiple defects. However, for preliminary comparison, this approach works well.

To organize the comparison, we collect a suite of test files for each type of defects. Each test file is targeting a specific circumstances of the defect. For instance, division by zero may be harder to detect for some tools when numeric type casting is involved, so we may add a test file for that scenario.

Each test file, for each supported programming language, presents two similar versions of a program—one good and one bad. The versions of the programs can be thought of as “before” and “after” fixing the corresponding defect. The versions in different languages are expected to be equivalent, at least from a software engineer’s perspective.

We implement test files as YAML documents with the following structure:

```yaml
title: # Title
description: >
  # Detailed description
features: # a list of tags
bad:
  source.cpp:
    # bad C++ program
  test.eo:
    # bad EO program
```

Analyzer	TP	TN	FP	FN	ERR	Accuracy	Precision	Recall	F1 score
Polystat (EO)	42	64	7	29	2	73.6%	85.7%	59.2%	70.0%
Polystat (Java)	0	56	0	56	32	38.9%	0.0%	0.0%	0.0%
Clang-Tidy	9	72	0	63	0	56.2%	100.0%	12.5%	22.2%
SVF	0	72	0	72	0	50.0%	0.0%	0.0%	0.0%
Cppcheck	6	72	0	66	0	54.2%	100.0%	8.3%	15.4%

Table 1. The comparison of performance metrics for Polystat and other static analyzers

```
good:
  source.cpp:
    # good C++ program
  test.eo:
    # good EO program
```

They are then used by automatic continuous integration scripts, to evaluate static analyzers whenever the benchmark suite repository on GitHub is updated.

2.2 Metrics

Assuming each tool is evaluated using one programming language, every test file contains essentially two programs. Running a tool on an a program leads to one of the following possible outcomes: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), or Error (ERR). Evaluating each tool on a collection of test files, we accumulate the following metrics:

Total count per type of outcome:

$$\text{Total} = TP + TN + FP + FN + ERR.$$

"Accuracy" as a ratio of TP+TN outcomes to the total number of test programs; this metric helps understand how good a tool is at predicting the presence of a defect:

$$\text{Accuracy} = \frac{TP + TN}{\text{Total}}.$$

"Precision" as a ratio of TP to the total number of TP+FP outcomes (predicted positives); this metric helps us understand how "useful" are positive detections of a defect in a program by a tool:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

"Recall" as a ratio of TP to the total number of TP+FN programs (actual positives); this metric helps us understand how well are actual defects detected by a tool:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

"F1 Score" as a harmonic mean of Precision and Recall; this metric is commonly used for preliminary comparison of tools, as high F1 score indicates that both Precision and Recall are good:

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Executing the overall benchmark produces an automated report, consisting of three parts. All metrics are grouped by the type of defect and the tool, forming the "statistics table." A detailed account of specific output for each tool run of every test file is recorded in a "details table." In this report we present only the statistics table, leaving the detailed parts of the report in the Appendix.

2.3 From C++ to EO

To properly compare Polystat with other analyzers, we need a way to provide equivalent programs in C++ and EO. While `c2eo`[1], a C/C++ to EO translator, is under development, it is not ready to be used for the purposes of this benchmark. In particular, many code examples used in our test files cannot be translated by the tool at the moment of writing.

To facilitate comparison, we translate C++ programs to EO manually. Some translation techniques have been discussed by Bugayenko (2021c) and Kudasov, Shilov, et al. (2021, Section 2). We apply those techniques manually, following these general principles: a) classes are translated to objects, capable of generating new objects; b) inheritance is approximated with decoration; c) `return` value is assigned to `@` attribute of a method.

While manual translation has its limitations, we believe that our test files are representative of Polystat's desired capabilities.

2.4 Types of defects

In this report, we focus on four types of defects. The first one, "division by zero," is a basic type of defect

[1]For more details see https://github.com/polystat/c2eo

supported by many tools. The other three are OOP-specific defects belonging to a category of fragile base class problems (Mikhajlov et al., 1998).

- `division-by-zero`: For this type of defect, we are looking for erroneous or at least fragile code, that may lead to a divide by zero error. Polystat detects this defect using an approach called "find-a-reverse" described by Bugayenko (2021b). Clang-Tidy supports detection of this defect via Clang Static Analyzer module `core.DivideZero`[2]. Support for this defect is also mentioned in Cppcheck, among other[3] defects.
- `mutual-recursion`: Here, we are interested in mutual recursion that occurs as a result of dynamic dispatch (virtual method overriding in subclasses). This problem is described for class-based OOP languages by Mikhajlov et al. (1998, Section 3.1) and for elegant objects by Kudasov, Shilov, et al. (2021, Section 3.2). Polystat supports this type of defect via `odin` module. Other analyzers do not claim to support this.
- `unjustified-assumption`: Here, we are interested in unjustified assumptions that occurs as a result of dynamic dispatch (virtual method overriding in subclasses). This problem is described for class-based OOP languages by Mikhajlov et al. (1998, Section 3.3) and for elegant objects by Kudasov and Sim (2022). Polystat supports this type of defect via `odin` module. Other analyzers do not claim to support this.
- `direct-state-access`: Here, we are interested in access (read/write) to the base class state without getters and setters methods. This problem is described for class-based OOP languages by Mikhajlov et al. (1998, Section 3.4). Polystat supports this type of defect via `odin` module. Other analyzers do not claim to support this.

In future versions of Polystat more types of bugs will be detectable.

2.5 Analyzers for comparison

The main criteria for choosing analyzers for comparison with Polystat were the availability and ease of integration. The selected analyzers are briefly described below:

- **Clang-Tidy** is a clang-based tool that is capable to detect various defects in C++ programs. In particular, Clang-Tidy includes C and C++ checkers from Clang Static Analyzer project.
- **Cppcheck** is an open source static code analysis tool for the C and C++ programming languages. It is versatile, and can check non-standard code including various compiler extensions, inline assembly code, etc. Its internal preprocessor can handle includes, macros, and several preprocessor commands.
- **SVF** is a source code analysis tool that enables interprocedural dependence analysis for LLVM-based languages. SVF is able to perform pointer alias analysis, memory SSA form construction, value-flow tracking for program variables and memory error checking.

If future reports we may use more analyzers.

3 Results

Benchmarking was carried out on a set of 144 tests, which can be found in repository[4]. In the statistic table in the appendix, we can see a summary comparison of Polystat and other analyzer performance over four types of defects.

Our interpretation of the metrics is as follows:

1. For OOP-specific defects, only Polystat has TP results. That says about his ability to find such defects.
2. Analyzers that did not claim to support OOP-specific defects, given the absence of TP results, do not actually find them. However, since they do not have FPs for those cases, the accuracy is exactly 50%.
3. Polystat shows 100% Recall for the OOP-specific defects on our test suit, meaning that Polystat has successfully detected all bad programs in test files. This indicates its good ability to find such defects, however, because of some FNs, the overall Accuracy is not as high. The FNs come from programs with branching (such as `if` statements or `while` loops), and in its current form Polystat cannot properly understand whether the condition should be taken into account.
4. For the division-by-zero defect that is not specific to OOP programs, the Clang-Tidy showed

[2]For more details see https://clang.llvm.org/docs/analyzer/checkers.html

[3]For more details see https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/

[4]For more details see https://github.com/polystat/awesome-bugs

the highest Accuracy of 78%. Polistat is only 3% behind.

Overall, we see that Polystat shows comparable performance for the common type of defects (division by zero), while the detection of OOP-related defects among the considered analyzers is supported only by Polystat.

4 Discussion and Summary

Our approach to benchmarking static analyzers has several limitations that we would like to address in the future work.

First, we would like to understand better, whether the tool correctly recognizes the type and location of the defect. However, this is technically difficult. In particular, we tried the following approaches:

1. Checking the reported line at which the defect was detected. Here, different analyzer may point out the defect at different lines in the source program.
2. Checking the pattern of defect messages. Unfortunately, the actual messages are not standardized, so extra analysis is required to compare output of different analyzer tools;
3. Check the defect/error code of the tool. Some analyzers do not report defect codes. Some analyzers use their internal code system, which requires mapping to some standardized map.

We also notice that while some of the defects (such as division by zero, or specific security vulnerabilities) are more or less standard across static analysis tools, OOP-specific defects are not standardized. To improve benchmarks for OOP static analysis, we suggest analysis and standardization of the common defects and anti-patterns in OOP programs.

Our benchmark test suit allows comparing static analyzers in a few specific scenarios. However, for the sake of completeness we think it is important to expand the test set in the future.

5 Conclusion

In this report, we have presented a basic methodology for comparing static analysis tools, a translation approach for comparing C++ tools with EO tools, and benchmark results comparing Polystat and other popular open source analyzers over test suit covering four types of defects: division by zero, unanticipated mutual recursion in subclasses, unjustified assumptions in subclasses, and direct access to the base class state.

We have shown that Polystat offers comparable performance for basic defects like division by zero, and outperforms others for detecting OOP-specific defects.

References

Bugayenko, Yegor (2021a). *EOLANG and phi-calculus*. arXiv: 2111.13384 [cs.PL].

– (2021b). "Finding a Reverse (FaR): Bug Detection in Object-Oriented Programs". In.

– (2021c). *Reducing Programs to Objects*. arXiv: 2112.11988 [cs.PL].

Kudasov, Nikolai, Nikolay Shilov, et al. (2021). "Detecting unanticipated mutual recursion using Elegant Objects representation of object-oriented programs". In.

Kudasov, Nikolai and Violetta Sim (2022). "Detecting unjustified assumptions in subclasses via elegant objects representation". In.

Mikhajlov, Leonid et al. (1998). "A Study of The Fragile Base Class Problem". In: *Proceedings of the 12th European Conference on Object-Oriented Programming*. ECCOP '98. Berlin, Heidelberg: Springer-Verlag, pp. 355–382. isbn: 3540647376.

A Statistic

Table 2 demonstrates detailed statistics collected for each static analyzer.

Analyzer	Defect title	TP	TN	FP	FN	ERR	Accuracy	Precision	Recall	F1 score
Polystat (EO)	division-by-zero	10	14	2	6	0	75.0%	83.3%	62.5%	71.4%
Polystat (EO)	mutual-recursion	26	22	4	0	0	92.3%	86.7%	100.0%	92.9%
Polystat (EO)	unjustified-assumption	6	7	1	2	2	72.2%	85.7%	75.0%	80.0%
Polystat (EO)	direct-state-access	0	21	0	21	0	50.0%	0.0%	0.0%	0.0%
Polystat (EO)	All	42	64	7	29	2	73.6%	85.7%	59.2%	70.0%
Polystat (Java)	division-by-zero	0	0	0	0	32	0.0%	0.0%	0.0%	0.0%
Polystat (Java)	mutual-recursion	0	26	0	26	0	50.0%	0.0%	0.0%	0.0%
Polystat (Java)	unjustified-assumption	0	9	0	9	0	50.0%	0.0%	0.0%	0.0%
Polystat (Java)	direct-state-access	0	21	0	21	0	50.0%	0.0%	0.0%	0.0%
Polystat (Java)	All	0	56	0	56	32	38.9%	0.0%	0.0%	0.0%
Clang-Tidy	division-by-zero	9	16	0	7	0	78.1%	100.0%	56.2%	72.0%
Clang-Tidy	mutual-recursion	0	26	0	26	0	50.0%	0.0%	0.0%	0.0%
Clang-Tidy	unjustified-assumption	0	9	0	9	0	50.0%	0.0%	0.0%	0.0%
Clang-Tidy	direct-state-access	0	21	0	21	0	50.0%	0.0%	0.0%	0.0%
Clang-Tidy	All	9	72	0	63	0	56.2%	100.0%	12.5%	22.2%
SVF	division-by-zero	0	16	0	16	0	50.0%	0.0%	0.0%	0.0%
SVF	mutual-recursion	0	26	0	26	0	50.0%	0.0%	0.0%	0.0%
SVF	unjustified-assumption	0	9	0	9	0	50.0%	0.0%	0.0%	0.0%
SVF	direct-state-access	0	21	0	21	0	50.0%	0.0%	0.0%	0.0%
SVF	All	0	72	0	72	0	50.0%	0.0%	0.0%	0.0%
Cppcheck	division-by-zero	6	16	0	10	0	68.8%	100.0%	37.5%	54.5%
Cppcheck	mutual-recursion	0	26	0	26	0	50.0%	0.0%	0.0%	0.0%
Cppcheck	unjustified-assumption	0	9	0	9	0	50.0%	0.0%	0.0%	0.0%
Cppcheck	direct-state-access	0	21	0	21	0	50.0%	0.0%	0.0%	0.0%
Cppcheck	All	6	72	0	66	0	54.2%	100.0%	8.3%	15.4%

Table 2. Each analyzer has received the same set of tests and then its key metrics, which are explained in Section 2.2, have been collected: the left column includes the name of the analyzer tested, the next one is the type of defect as explained in Section 2.4, then one column per each metric.

B Details

Table 3 demonstrates detailed results for each test by each analyzer.

Defect type	File	Polystat (EO)	Polystat (Java)	Clang-Tidy	SVF	Cppcheck
direct state access	read-in-calculation-chain-2.yml	PG	PG	PG	PG	PG
direct state access	read-in-calculation-chain.yml	PG	PG	PG	PG	PG
direct state access	read-in-factory.yml	PG	PG	PG	PG	PG
direct state access	read-in-inheritance-chain.yml	PG	PG	PG	PG	PG
direct state access	read-inheritance-chain-nested-1.yml	PG	PG	PG	PG	PG
direct state access	read-inheritance-chain-nested-2.yml	PG	PG	PG	PG	PG
direct state access	read-nested-class-1.yml	PG	PG	PG	PG	PG
direct state access	read-nested-class-2.yml	PG	PG	PG	PG	PG
direct state access	read-non-primitive-data.yml	PG	PG	PG	PG	PG
direct state access	read-simple-1.yml	PG	PG	PG	PG	PG
direct state access	read-simple-2.yml	PG	PG	PG	PG	PG
direct state access	write-in-factory.yml	PG	PG	PG	PG	PG
direct state access	write-in-inheritance-chain.yml	PG	PG	PG	PG	PG
direct state access	write-inheritance-chain-nested-1.yml	PG	PG	PG	PG	PG
direct state access	write-inheritance-chain-nested-2.yml	PG	PG	PG	PG	PG
direct state access	write-nested-class-1.yml	PG	PG	PG	PG	PG
direct state access	write-nested-class-2.yml	PG	PG	PG	PG	PG
direct state access	write-non-primitive-data.yml	PG	PG	PG	PG	PG
direct state access	write-simple-1.yml	PG	PG	PG	PG	PG
direct state access	write-simple-2.yml	PG	PG	PG	PG	PG
direct state access	write-through-another-method.yml	PG	PG	PG	PG	PG
division by zero	array-size-in-method.yml	PG	E	PG	PG	P
division by zero	array-size.yml	P	E	PG	PG	PG
division by zero	branching.yml	P	E	P	PG	PG
division by zero	calculation-chain.yml	PB	E	P	PG	P
division by zero	cast-bool-int.yml	P	E	P	PG	P
division by zero	cast-float-int.yml	PG	E	PG	PG	P
division by zero	default-value-in-client.yml	PG	E	PG	PG	PG
division by zero	default-value-in-method.yml	PG	E	PG	PG	PG
division by zero	expression-in-divisor1.yml	P	E	P	PG	PG
division by zero	expression-in-divisor2.yml	P	E	P	PG	P
division by zero	function-in-divisor.yml	PB	E	P	PG	PG
division by zero	nested-branching.yml	P	E	P	PG	PG
division by zero	parameter-in-divisor.yml	P	E	P	PG	PG

division by zero	random.yml	PG	E	PG	PG	PG
division by zero	recursive-call.yml	PG	E	PG	PG	PG
division by zero	zero-in-divisor.yml	P	E	P	PG	P
mutual recursion	in-chain-of-calls.yml	P	PG	PG	PG	PG
mutual recursion	in-factory.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-1.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-2.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-3.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-4.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-1.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-2.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-3.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-4.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-base-1.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-base-2.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-base-3.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-base-4.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-derived-1.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-derived-2.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-derived-3.yml	P	PG	PG	PG	PG
mutual recursion	in-inheritance-chain-nested-derived-4.yml	P	PG	PG	PG	PG
mutual recursion	nested-base.yml	P	PG	PG	PG	PG
mutual recursion	nested-derived.yml	P	PG	PG	PG	PG
mutual recursion	nested.yml	P	PG	PG	PG	PG
mutual recursion	with-if-branching1.yml	PB	PG	PG	PG	PG
mutual recursion	with-if-branching2.yml	PB	PG	PG	PG	PG
mutual recursion	with-if-branching3.yml	PB	PG	PG	PG	PG
mutual recursion	with-random-if-branching.yml	PB	PG	PG	PG	PG
mutual recursion	mutual-recursion.yml	P	PG	PG	PG	PG
unjustified assumption	calls-chain.yml	E	PG	PG	PG	PG
unjustified assumption	fragile-baseclass-example.yml	PG	PG	PG	PG	PG
unjustified assumption	in-factory.yml	P	PG	PG	PG	PG
unjustified assumption	in-recursion.yml	PB	PG	PG	PG	PG
unjustified assumption	inheritance-chain1.yml	P	PG	PG	PG	PG
unjustified assumption	inheritance-chain2.yml	P	PG	PG	PG	PG
unjustified assumption	inheritance-chain3.yml	P	PG	PG	PG	PG
unjustified assumption	multiple-subclasses-with-defect.yml	PG	PG	PG	PG	PG
unjustified assumption	unjustified-assumption.yml	P	PG	PG	PG	PG

Table 3. Here, “P” means pass for Bad and Good cases,“PG” means pass for Good cases,“PB” means pass for Bad cases,“F” means fails for Bad and Good cases,and “E” means errors/exceptions during analysis