Program Analysis In Practice

Shrey

Why should you listen to me?

Experience - Uber Research

Interned at the **programming systems group** at Uber this summer

Part of the **software reliability** team - group of PhDs specializing in PL and AI

Developed program analysis tools for **automating code reviews** and **false positive elimination**!

Experience - Microsoft Research

Research Fellow at MSR India from 2021 to 2023 (before joining CMU)

Part of the **Cloud Reliability** team for Microsoft Azure

Developed on **Static** and **Dynamic Resource Leak Detection** tools for Cloud Services

Experience - Citrix Systems

Worked as a **software developer** in the VPN solutions team from 2020 to 2021

Reviewed many comments from static analysis tools like **sonarqube** and **coverity** before pushing code to production :')

Improved existing dynamic bug finding techniques (testing) for faster execution?

What Will I Talk About Today?

My experience at Uber developing/improving program analysis tools:

1. Static Program Analysis: **NullAway** and **NilAway**

2. Program Reasoning: **uReview**

While the talk is focused on my work from Uber, the lessons and discussion points are generally applicable to all program analysis techniques in practice!

NullAway and NilAway

Simple Real World Example

```
L109.  type Conn interface {
...        ...
L128.      RemoteAddr
...        ...
L164.  }
...
       // struct net
L166.  type conn stru
L167.      fd *netFD
L168.  }
...
L223.  func (c *conn)
L224.      if !c.ok
L225.          ret
L226.      }
L227.      return c
L228.  }
```

4 src/net/http/server.go

```
@@ -1856,7 +1856,9 @@ func isCommonNetReadError(err error) bool {
1856  1856
1857  1857   // Serve a new connection.
1858  1858   func (c *conn) serve(ctx context.Context) {
1859        -          c.remoteAddr = c.rwc.RemoteAddr().String()
      1859  +          if ra := c.rwc.RemoteAddr(); ra != nil {
      1860  +                  c.remoteAddr = ra.String()
      1861  +          }
1860  1862             ctx = context.WithValue(ctx, LocalAddrContextKey, c.rwc.LocalAddr())
1861  1863             var inFlightResponse *response
1862  1864             defer func() {
```

panic: runtime error: invalid memory address or nil pointer dereference

Crashes In Production Cost $$$

- App and service crashes can cause significant problems to users, such as preventing riders from requesting a trip in a timely manner or drivers from accepting rides.

- **Null Pointer Exceptions,** which occur when a null pointer is dereferenced in Java, are a frequent cause of crashes in Uber's android apps.

- Similarly, Go services at Uber have witnessed several runtime errors in production because of **nil panics**, with effects ranging from incorrect program behavior to app outages.

Solution: Static Analysis Tools

- Uber uses monorepos meaning all code in a specific language is stored inside a single repository.

- This makes static analysis a very attractive option since you just need to ensure that the entire repository is free of any null pointer exceptions or nil panics!

- They developed two tools:
  - NullAway for Java -> Annotation-based type checking for NPE's
  - NilAway for Go -> Type checking for nil panics

NullAway For Java

Built using the **Java Checker Framework** for pluggable type checking.

Question: What is annotation-based program analysis and how is it related to type checking? How is it different from traditional interprocedural analysis?

NullAway for Java

One of the simplest annotation-based analysis. Makes use of only two annotations:

- **@NonNull:** A type that can never be null
- **@Nullable:** A type can may or may not be null

The checker checks for two invariants:

1. No expression of **@Nullable** type is ever assigned to a location of **@NonNull** type.
2. No expression of **@Nullable** type is ever dereferenced.

NullAway In Action

Let's look at their playground: [EISOP Checker Framework Live Demo]

Results of Deploying NullAway at Uber

The tool was successful deployed on all of Uber's Java code.

NullAway identified many potential NPE bugs that were fixed by the developers leading to significant reduction in app NPE's logged.

Since it is very hard to annotate the entire existing code base (millions of lines of Java code), the tool made default assumptions rendering it neither sound nor complete.

Can we do better?

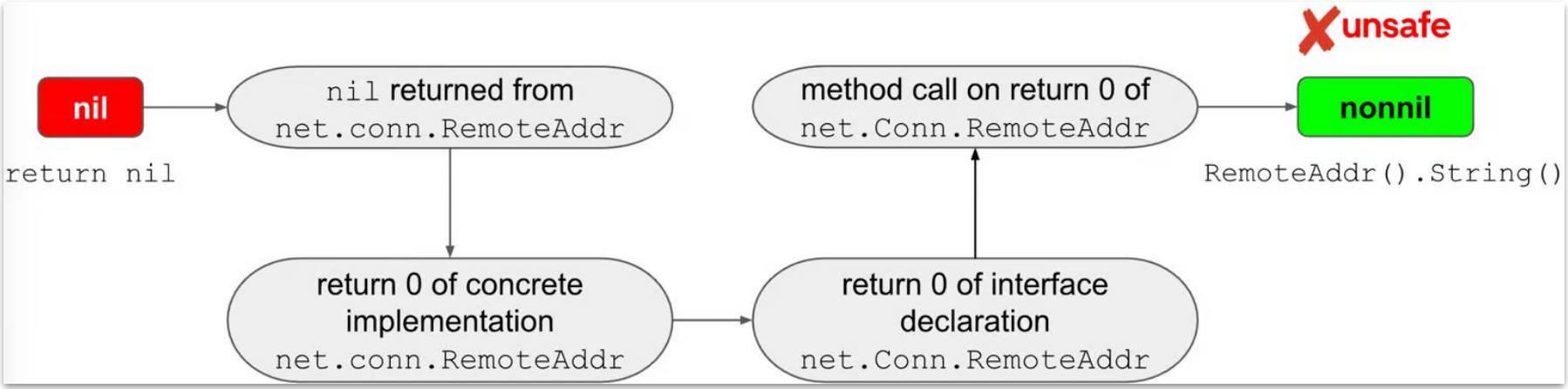
NilAway for Go

Main idea: Get rid of annotations and collect typing constraints automatically looking for contradictions.

An example of a **nilable** constraint is return x, where x is an uninitialized pointer, while the dereference, *x, is an example of a **nonnil** constraint.

A contradiction occurs when for a program site S, **nilable(S) ^ nonnil(S)** is discovered to be true.

NilAway for Go




```
go/src/net/http/server.go:1859:17: Potential nil panic detected. Observed nil flow from
source to dereference point:
    -> net/net.go:225:10: `nil` returned from `conn.RemoteAddr()`
    -> net/net.go:128:2: returned from interface method `Conn.RemoteAddr`
    -> net/http/server.go:1859:17: `c.rwc.RemoteAddr()` called `String()`
```

Results of Deploying NilAway at Uber

The tool was successful deployed on all of Uber's Go code (100 million+ lines of code).

NilAway has reported over **10,000 nil panic alerts** till date and has been in production for almost 2 years now.

The tool has a **precision of 60%** meaning ~6,000 nil panic alerts have been addressed by developers at Uber so far!

Questions

1. What according to you is an acceptable level of precision for a program analysis tool and why? How would you even measure the precision of the tool? Feel free to take concrete examples and answer!

2. What are the reasons for imprecision in static analysis tools? Can you think of ways to improve their precision? Feel free to either use NullAway/NilAway as an example or discuss any analysis of your choice!

uReview: Scalable & Trustworthy GenAI for Code Reviews

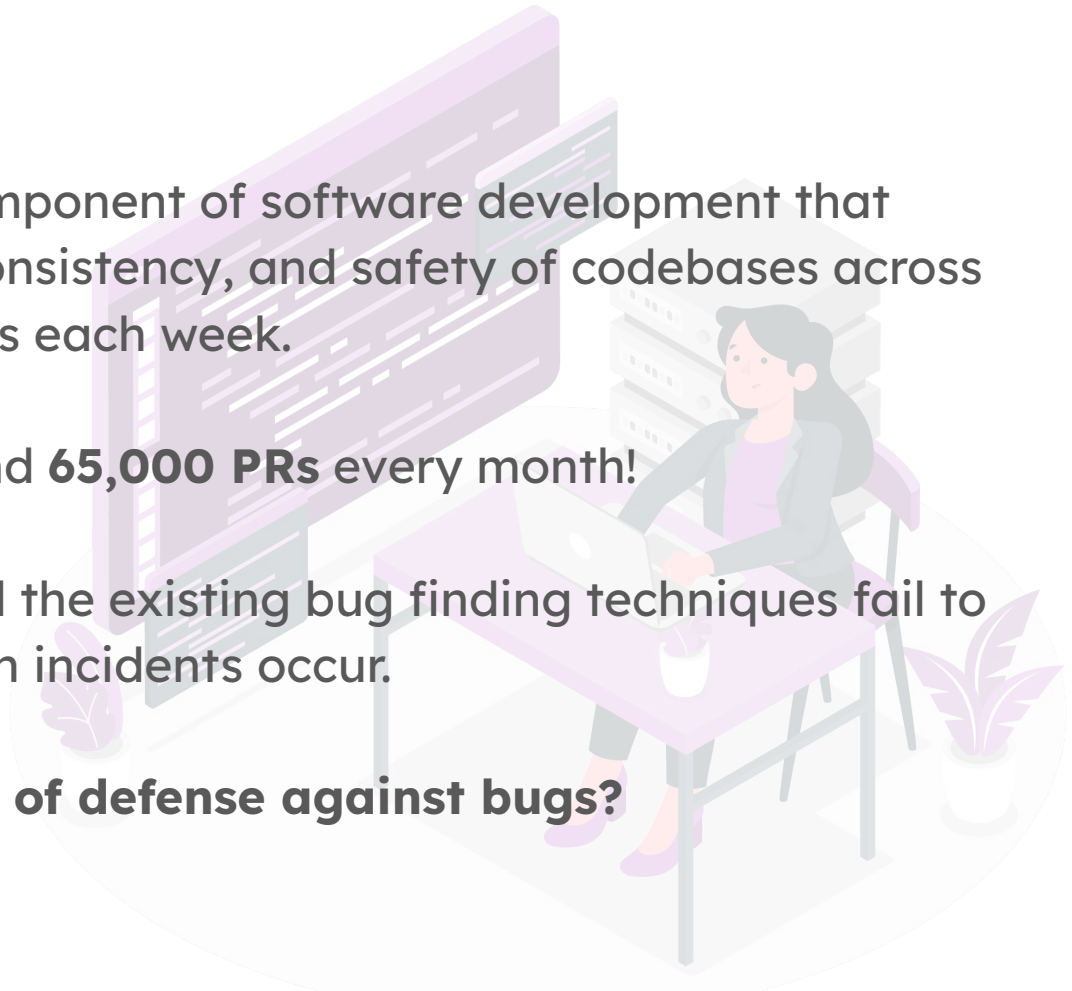
Motivation

- **Code reviews** are a core component of software development that help ensure the reliability, consistency, and safety of codebases across tens of thousands of changes each week.

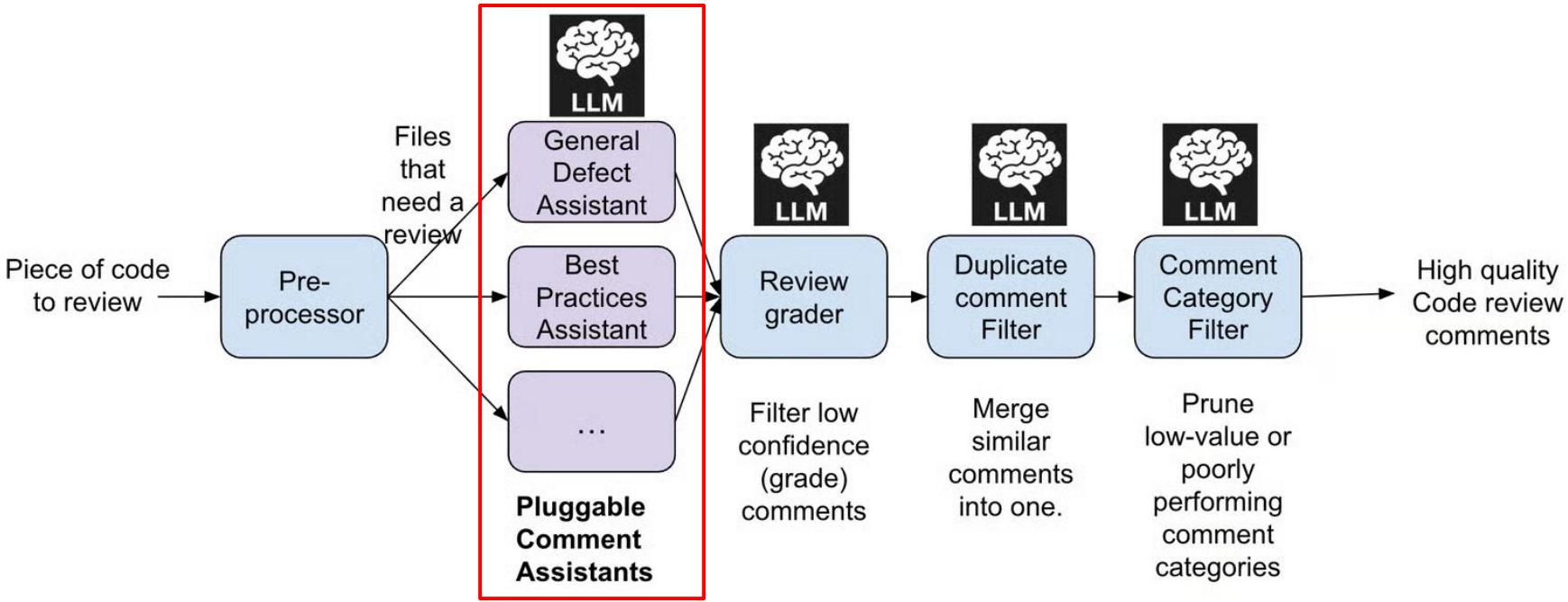
- Uber's monorepos see around **65,000 PRs** every month!

- When humans reviewers and the existing bug finding techniques fail to detect code bugs, production incidents occur.

- **Why not add another layer of defense against bugs?**



High-Level Architecture

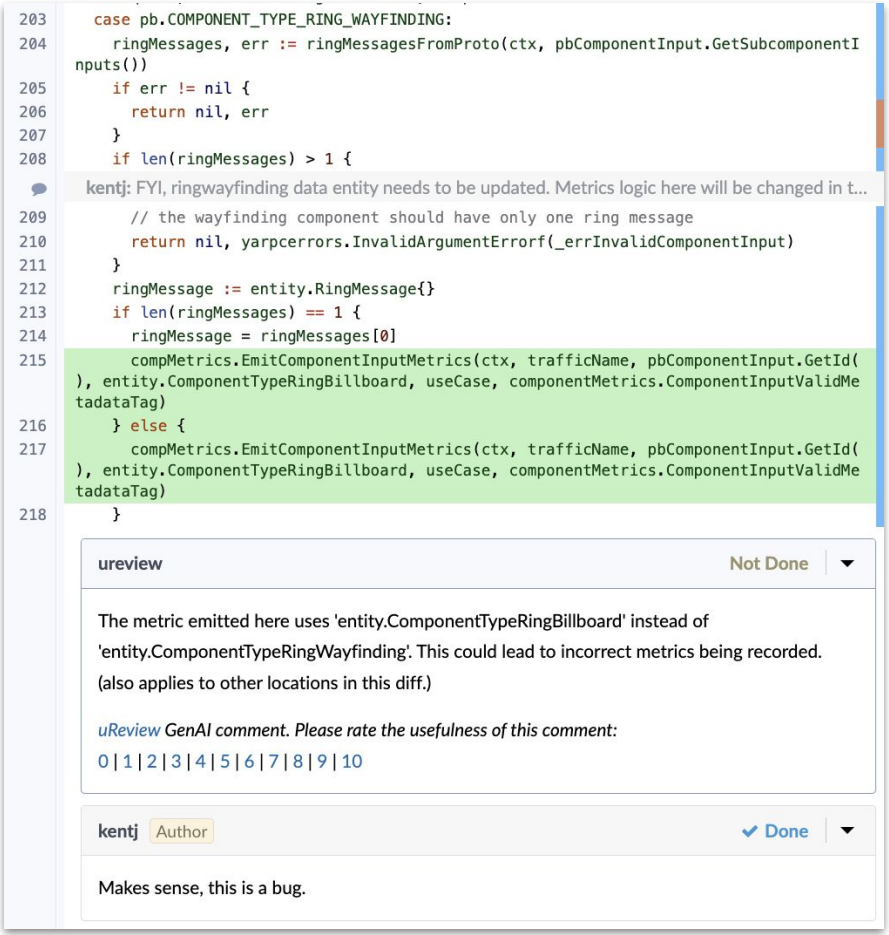


uReview Comments

All the PRs raised are usually reviewed in < 5mins

Comments are posted similar to how a human reviewer would

Developers can provide feedback on comments to help improve the tool!

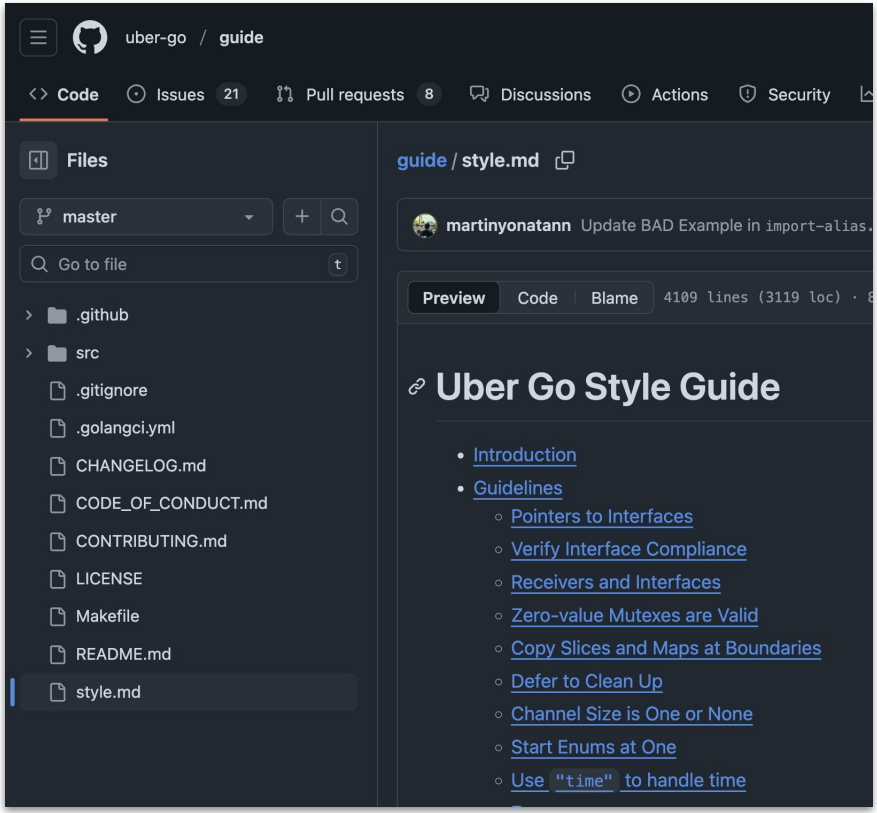


General Defects & Best Practices Bots

GDB is a simple prompt based bot that asks the LLM to review code and look for bugs

BPB is a rule-based bot that asks the LLM to reason about rules and code context to find violations

Many other bots are pluggable…



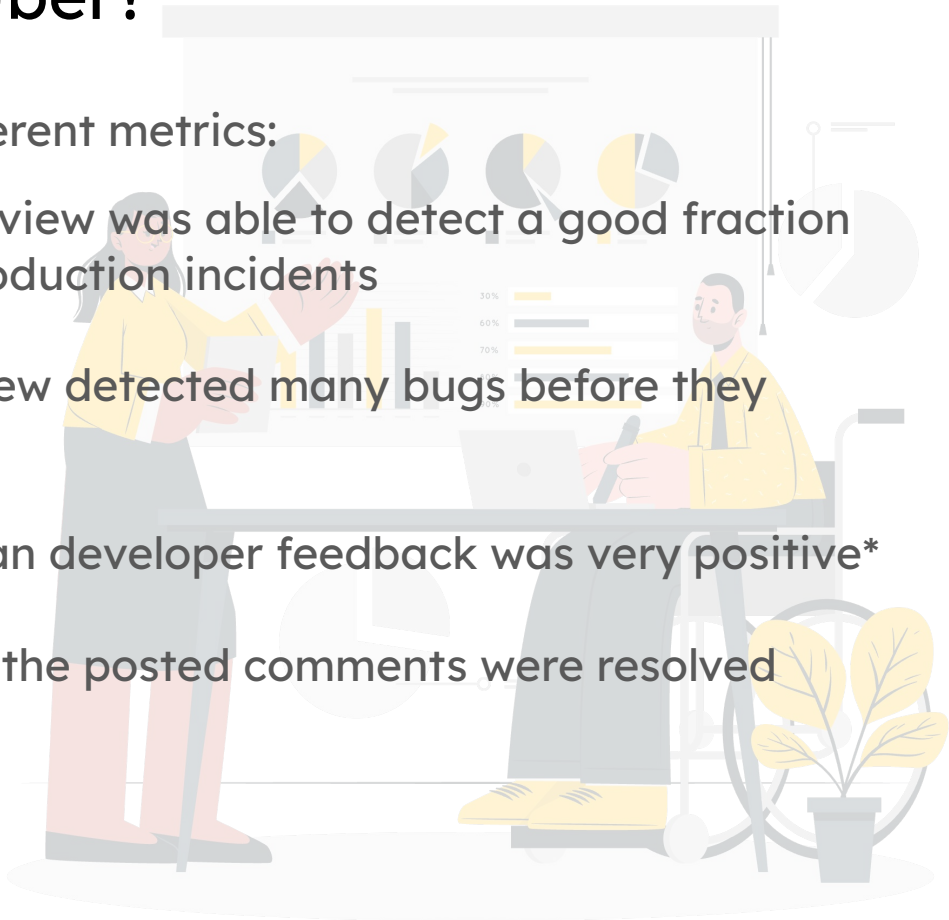
How useful is uReview at Uber?

1. If you were developing a program analysis tool in the industry, how would you measure the usefulness and impact of your tool in the company?

How useful is uReview at Uber?

We establish usefulness using a few different metrics:

- **Retrospective Detection Rate:** uReview was able to detect a good fraction of historic bugs that lead to past production incidents
- **Preventable Incident Count:** uReview detected many bugs before they reached production*
- **Developer Satisfaction Rate:** Median developer feedback was very positive*
- **Comment Addressal Rate:** **~65%** of the posted comments were resolved



Questions

1. Why might developers ignore uReview's bug comments, even when they have been shown to be useful? This phenomenon is not unique to uReview—consider why similar challenges arise with other program analysis tools too.

2. Imagine you are the project lead for uReview. What are some of the lessons learnt from our discussion about problems? How would you solve these problems?

Improving the performance of the BPB

Goal: Improve the usefulness of the BPB in Uber.

Main Problems: Missed violations and hallucinations

Why only BPB?

- Limited time for the project (~4 weeks)
- BPB does not detect bugs so it's harder to convince developers that the comments posted need to be addressed

Solution: BPB V2.0

Spend more **$$$** and perform **focused reasoning** on each rule for every file in the PR

Curated best practice rules from developers with both positive and negative examples for each rule

Implemented a **self-improving RAG-based post processing** step to filter out less valuable comments generated

Research Questions

RQ1: What were the total number of comments posted by the new BPB?

RQ2: What was the quality of the new comments being posted?

RQ3: What was the addressal rate for these new set of comments?

RQ4: What was the developer feedback for the new BPB?

Quantity and Quality of Comments

RQ1: What were the total number of comments posted by the new BPB?

The BPB V2.0 posted almost the same number comments in **2 weeks** of deployment as the BPB V1.0 in its **6 months** of deployment

RQ2: What was the quality of the new comments being posted?

Evaluated the results of the new bot on a manually labelled dataset of internal PRs. Observed an **3.5x improvement** in both precision and recall!

Developer Experience

RQ3: What was the addressal rate for these new set of comments?

The comment addressal rate went up by **15%** for the new bot. We expect it to go even higher as the bot collects more data and filters comments!

RQ4: What was the developer feedback for the new BPB?

The average **comment score increased** slightly and the feedback was positive. The important point is the new bot facilitates **collection of a lot more data/feedback** due to its volume of comments.

What’s one key takeaway for me after working with program analysis for all these years?

Key Takeaway

Success of a program analysis tool is defined by its **impact** and not **effectiveness**.

Impact = Effectiveness * Applicability * Trust

Impact measures the ability of a tool to save $$$ for a company and that's the only thing that matters at the end of the day!

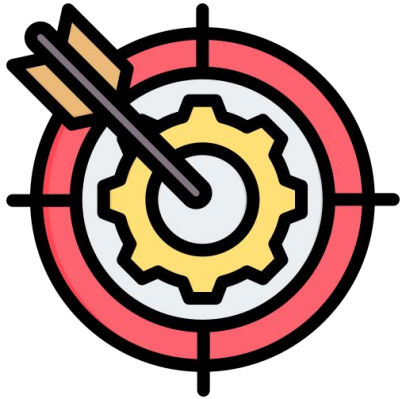
Effectiveness

Usually refers to the precision and recall of the program analysis tool.

Academia usually focuses on this one aspect* but this is not the only metric to consider when building tools.

Sometimes effectiveness is also measured in other ways:

- Time saved for developers
- Mean time between failures
- Cost of preventable incidents...



Applicability

Law of bug finding: You cannot find bugs in the code you don't analyze.

Tools have to be very applicable to increase the amount of code analyzed.

Many factors affect applicability; we should control all that we can!

Example: RLC# vs NilAway

Trust

Developer trust is of utmost importance since they are responsible for resolving all the alerts raised.

If the output of the tool is noisy, developers start treating all alerts as false positives. This affects impact.

No matter how important the bug type, completeness is more important than soundness.

Further Reading

Many interesting insights and a overall fun read!!

DOI:10.1145/1646353.1646374

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later

Using Static Analysis to Find Bugs in the Real World