The Bug That Cried Wolf

Edward Chen
UC San Diego

Arvind Saripalli
UC San Diego

Shun-Wen Yu
UC San Diego

Abstract

With the prevalent use of static analysis checkers and the rise of new tools of this kind comes a necessity to ensure the effectiveness and reliability of these tools. In order to accomplish this task, we developed a framework, Breezy, which allows automated bug tracking from the inception to the resolution of a bug within any given Github repository. Our current tool utilizes Clang, Cppcheck, and Infer, 3 commonly used static analysis tools deployed in industry, but can easily be extended to incorporate any additional checkers. For our experiments, we analyzed a few small scale C++ based repositories and the Mozilla Gecko-Dev repository. Our analysis indicates we are able to successfully track a bug's lifespan, depending on the effectiveness of the static analysis tool. We also conducted analysis on the effectiveness of static analysis tools by comparing results of those tools against fixed bugs documented on Firefox's bug tracking platform, Bugzilla. For future endeavors, we plan on performing large scale data analysis on reported bugs and implement a ranking system to determine the false positive rate of a flagged bug.

Keywords static analysis, bug finding, bug tracking, bug ranking

1 Introduction

Vulnerabilities within a piece of software are often times hard to find and costly if not fixed promptly. Thus many static analysis tools have been created in order to make finding bugs easier. However, many of these static analysis tools suffer from having a high false positivity rate, reducing the urgency for developers to fix flagged bugs. In this paper, we present a framework, Breezy, for tracking a bug's lifespan from the inception to the resolution of a bug. Breezy applies static analysis tools on previous commits within a Github repository, saving any updates to previous or new bugs found. The data can be later used to reveal possible bugs to newly introduced code, or even decide how likely a bug is actually a true positive. This framework can eventually be extended to incorporate newly created static analysis tools and or even be integrated into systems that currently run static analysis tools.

2 Background Related Work

Static analysis tools are a commonly distributed bug detecting system that is widely used throughout both industry and academia. To our knowledge, there exists no other system that currently utilizes static analysis tools across Github repositories, specifically through past commits, or compares the end results of multiple static analysis tools.

Static Analysis Tools Background Static analysis tool research is far from a new concept [7] [10]. Previous research within the bug finding field indicates some promising results from utilizing our framework. A review from Wagner et. al. showed that bugs found through static analysis checkers may not necessarily be caught through testing frameworks, but a subset of these bugs still appear within bug reviews [13]. Therefore, bugs caught by tools may serve to be true security vulnerabilities.

Lessons from Industry Static analysis tools are notorious for having false positive rates when analyzing for bugs. Recent research shows that when deploying these static analysis tools across large scale systems at Google [4] [11] and Facebook [5], many of the developers lose faith in these tools due to the high number of probable but false bugs that these tools were detecting. Although changing the workflow of these tools to be more on-demand than overnight runs served some benefit in fixing more flagged bugs, static analysis tools are still associated with having a high false positive rate.

Reducing the number of False Positives Many approached have been taken in reducing the number of false positives from the static analysis tools. The primary approach is to utilize text and specification mining in order to predict whether a flagged bug is a true positive or not [9] [8]. These systems usually rely on the static analysis bug reports as input data and can be incorporated on top of existing static analysis detection systems.

Static Analysis Tools Used For this project, we chose to use Cppcheck, Infer, and Clang. Cppcheck is a lightweight static analysis tool focused solely on detecting bugs in C/C++ code. Cppcheck uses unsound flow sensitive analysis, sacrificing the accuracy of the checker in order to reduce the number of false positives. This tool has an extremely low entry bar with its ability to run on uncompiled C/C++ repositories and has the quickest turn-around time compared to Clang and Infer. [2] Infer is another static analysis tool created and widely deployed across the systems at Facebook. This static analysis tool focuses on code written in Java, C, C++, and Objective-C. There are two implementations of this tool: 1. Infer.SL which uses separation logic to detect bugs 2. Infer.AI which is a more generalized version of Infer focusing on modular analysis. For our project, we only tested the Infer.SL implementation of Infer, but our framework can support both implementations[3]. The last tool we used is Clang Static Analyzer (Clang), a commonly used static analysis tool for C, C++, and Objective-C code. This static analyzer "can detect dead code or memory leaks, but as a typical side effect they have false positives" [1].

3 Breezy Framework

Breezy is a framework that we developed to track bugs over time in repositories. Given a range of commits to analyze, it can determine when bugs are created and when they become resolved, as well as statistical trends for these bugs.

Breezy takes as input any repository as well as a specification for a static analysis tool to use. Currently, Clang, Cppcheck, and Infer are the static analysis checkers that are supported. Static analysis tools can be configured, mixed and matched so that Breezy users can analyze exactly the kinds of bugs that they are looking for in their repositories. For example, Breezy can be configured to work as a linter and analyze linting errors over time.

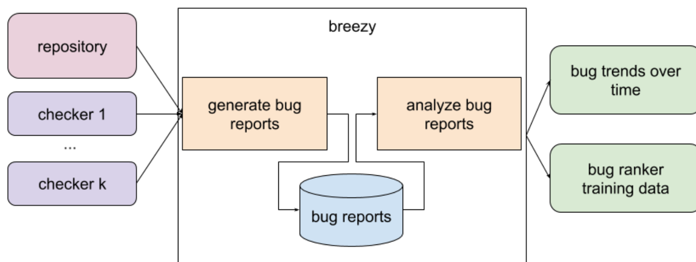

Figure 1. Breezy high level architecture

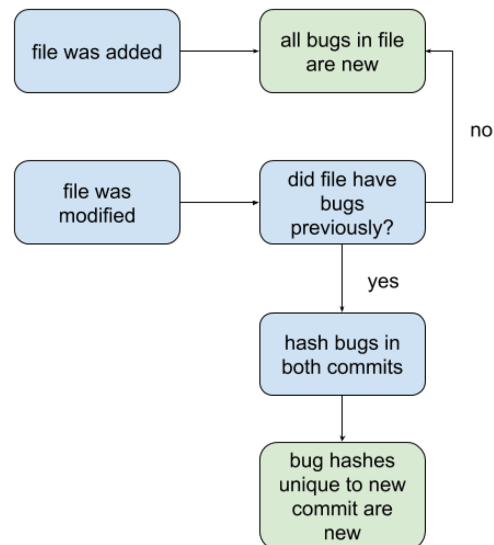

Figure 2. Determining which bugs were conceived in a commit

3.1 What is a bug?

We based our definition of a bug based on what we observed in the output of some commonly used static bug checkers. A bug in our system is defined by its

- file
- line(s) of code (LOC)
- buggy code
- type of bug
- description

We store some additional information about a bug such as the commit that it was found in. Note that the same bug can be found in multiple commits, so this is not a piece of information that can be used to readily identify the bug.

Additionally, a bug can move around in a file if the file is modified. So, across commits, the same bug can have different LOCs. Therefore when we hash bugs to distinguish them, we leave out the LOC and commit information.

3.2 Bug Reports

The basis for finding bugs in our system is to run a static bug checker on a window of commits of a specified repository. The output of the checker is parsed into a list of bugs. We store this list of bugs as a bug report data structure, a structured representation of the bug information for a single commit. We step through each commit in the window, and save the resulting bug report for each commit in the window.

The saved bug reports are used down the line to analyze bug trends over time in the given commit window.

3.3 Tracking bugs over time

3.3.1 Bug conception

As a base case, we treat all bugs in the starting commit of the commit window as having been conceived in that commit.

Given commit n, commit $n+1$ immediately following it, and bug reports for each commit, we would like to determine which bugs in commit $n+1$ are new. To do this, we iterate through each changed or added file between these commits. We check the bugs in these files as shown in Figure 2.

When a new file is added, all of its bugs are new. If a file is modified and it previously did not contain bugs, then all of its bugs are new. In other cases, we obtain hashes for bugs in each commit. Our hash is defined over all of the bugs attributes except for its location and commit. This is because bugs can move around in code and are persistent across commits, so those attributes do not identify a bug. Once we have the hashes, all of the hashes in the commit $n+1$ that are not in commit n are considered to be new bugs.

All of the new bugs found in commit $n+1$ are added to the pool of bugs being tracked over time.

3.3.2 Tracking a single bug over time

Given a single bug, we use the following approach (3) to determine when it is resolved in the commit window, if at all. From commit to commit, if the file that the bug is contained in was modified, then we examine the diff of the file. If the LOCs of the bug were part of the modification to the file, then we cross examine the bug reports from the commits to determine whether the modification resolved the bug or not. If the LOCs of the bug were not part of the modification, we use the diff to determine the new LOCs for the bug. If the file was renamed, then the file in the bug is adjusted accordingly.

3.3.3 Static Bug Checkers

Because our definition of a bug is a relatively simple abstraction, Breezy can use many different static checkers and potentially in combination with one another. In our own tests, we relied on Cppcheck, Infer, and Clang.

Adding support for new static checkers is relatively simple. The main code addition to support a new static checker is to write a parser for the output of the static checker to our bug class.

4 Using Breezy

4.1 mozilla/gecko-dev

As part of the evaluation of our bug tracking tool, we ran our tool on the Mozilla Firefox source code. The Mozilla Firefox source code

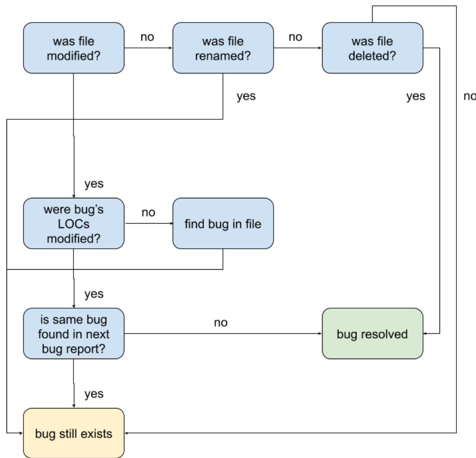

Figure 3. Determining whether a bug has been resolved in a commit

provides a large and highly bug-prone code base for us to perform analysis upon. Furthermore, bugs are flagged and resolved daily in a well-documented fashion through Bugzilla. Using information from Bugzilla, we can compare the types and location of bugs documented on Bugzilla against bugs we find using static checkers such as Clang.

4.2 Bugzilla

The open source gecko-dev repository uses a platform called Bugzilla to report, document, and review bugs as well as bug fixes. Bugs documented on Bugzilla contain information such as: product involved, component involved, resolution status, priority, severity, and other information. Bug reports typically include attachments that document steps to reproduce the bug, and/or relevant log files. For fixed bugs, the attachments also include git diffs for the files changed either through a text file or through a link to a code review entry on Phabricator, which is a code review platform used by gecko-dev.

4.3 Bugzilla statistics

We used Bugzilla to parse information about resolved bugs, which was facilitated by interfacing with Bugzilla's REST API. We looked at all resolved bugs of major/critical severity within the past two years, and parsed their attachments to obtain file changes associated with each bug fix. Using this information, we plotted the top twenty files changed with major/critical bug fixes in **Figure 5** and the top twenty top level directories involved in major/critical bug fixes in **Figure 4**. Both plots show the disproportionate amount of bug fixes involving the Javascript engine as well as the DOM (Document Object Model). We think this is because of the complexity of these files as well as the frequency of practical use in these respective repositories.

4.4 Clang on gecko-dev

We ran Clang on gecko-dev across multiple commits to compare Clang's static analysis data against documented bug reports on Bugzilla. By jumping at 100,000 commits at a time, we ran Clang on each of those commits to find the number of bugs caught by Clang at each point in time. This is plotted as a line graph in **Figure 6**.

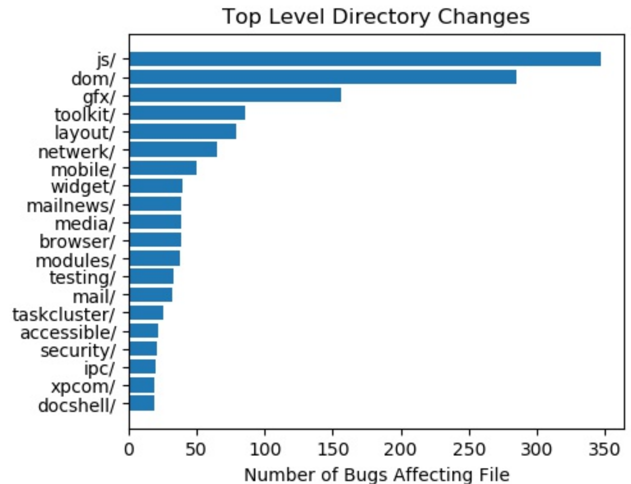

Figure 4. The top 20 directories involved in resolved major/critical bugs reported on Bugzilla

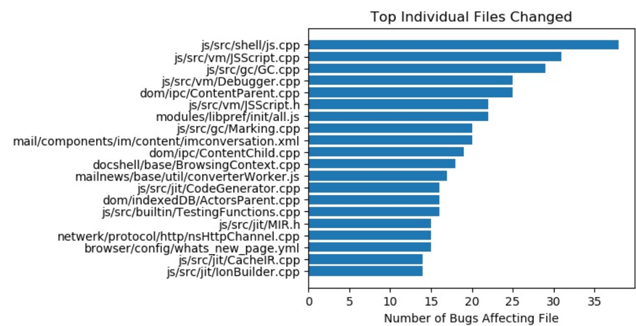

Figure 5. The top 20 files involved in resolved major/critical bugs reported on Bugzilla

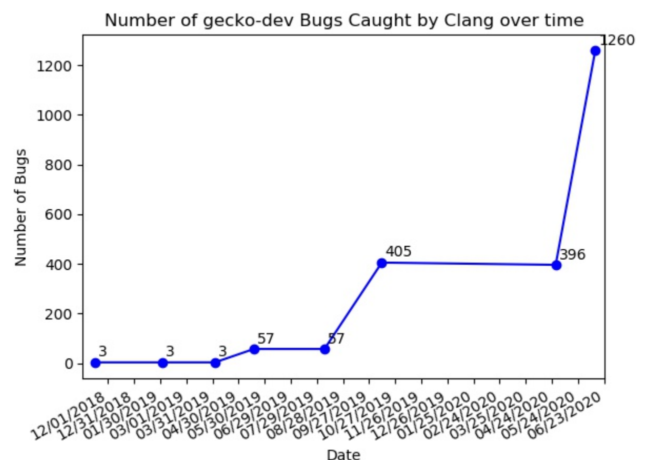

Figure 6. The number of gecko-dev bugs caught by Clang over time

We find that, naturally, the number of bugs increases as the code

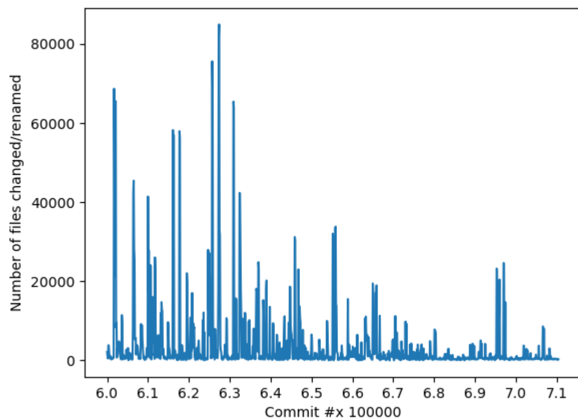

Figure 7. gecko-dev file changes over time

base becomes larger, but we were unable to account for the sudden leap from 396 bugs to 1260 in a little less than a two month period. Initially, we thought this may have had to do with a sudden increase in code changes during that period, but running git diffs across the entire lifespan of the repository showed that the lines of code change during that was not significant over that period. Our results of running git diff across the repository's lifespan can be seen in **Figure 7**.

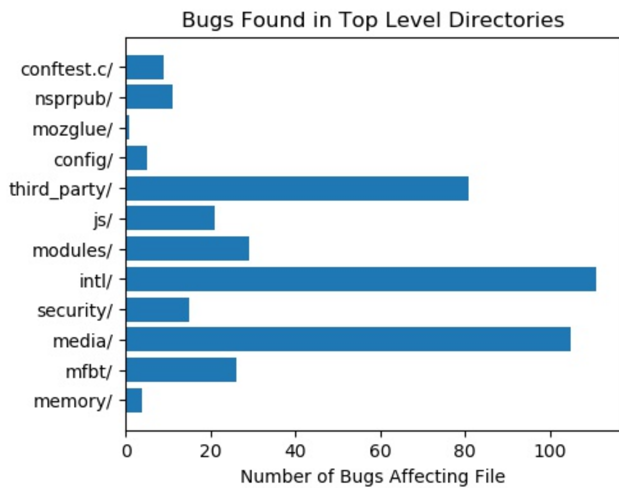

Figure 8. The frequency of bugs found in directories by Clang

4.5 Comparing clang output with Bugzilla

We used the clang reports over multiple commits to generate information about the number of bugs found in a given file (**Figure 9**) and the number of bugs found in a given directory (**Figure 8**) over time. To avoid duplicating the count for an unfixed bug across several commits, we hashed the (filename, description, code snippet, bug type, and severity) of each bug and counted the hashes of these bugs. However, this method only works as an approximation, as there were instances in which the same buggy code was repeated

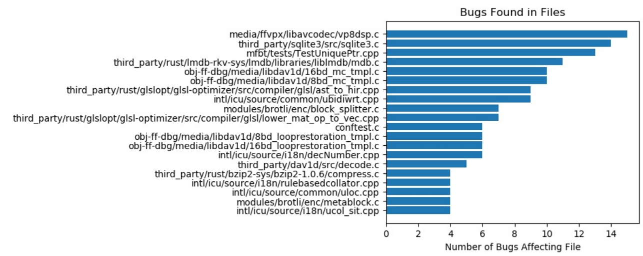

Figure 9. The frequency of bugs found in individual files by clang

in multiple locations within the same commit, and cases where a bug persisted in a later commit after a minor code change.

Comparing buggy files and directories found by clang against those documented on Bugzilla, we see a drastic difference. Clang found many bugs in directories involving localization, media handling, and third party software. Media handling involves a lot of bit-level operations that are notoriously difficult to get right, and third party software does not receive the same scrutiny as software developed for Firefox. We hypothesize that this difference is also accounted for by the fact that the bugs documented on Bugzilla are more often logical bugs whereas the bugs caught by clang are more low level memory-related bugs. Bugs clang caught involve less critical code paths, making them more difficult to be discovered through regular use.

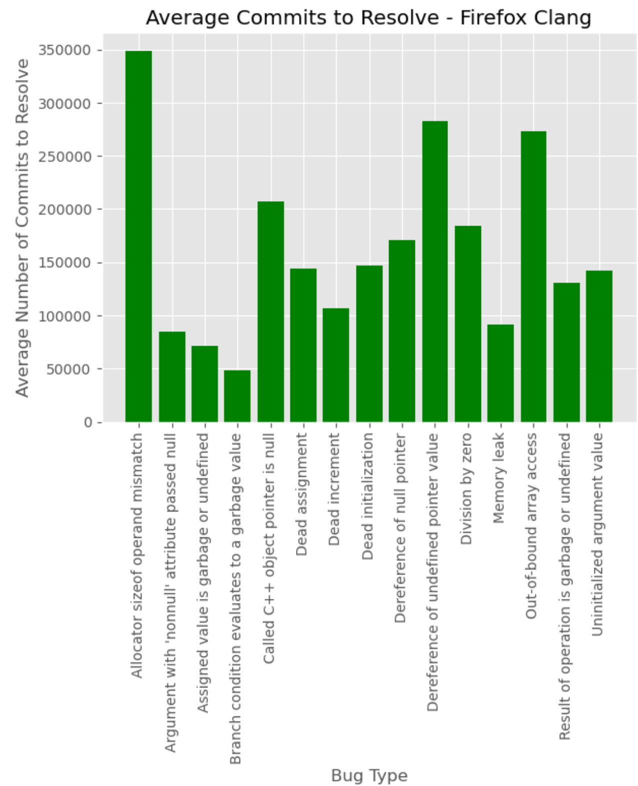

Figure 10. The average number of commits to resolve each type of bug

4.6 Using Breezy on Firefox with Clang

We ran Breezy on Firefox with Clang and for each bug that we found to be resolved at a later commit, we documented the time it took for each type of bug (as specified by Clang) to resolve. From this, we found that bugs involving "Allocator sizeof operand mismatch" bugs take, on average, the most commits to resolve. We speculate that this is due to the fact that errors involving the sizeof operand are typically harder to discover and often have no significant functional consequence. Otherwise, we believe the average number of commits it takes to resolve can serve as a indicator for the severity of a given bug type. Bugs that take less commits to resolve could mean that they are of more importance and bugs that take more commits to resolve may mean that they are less significant.

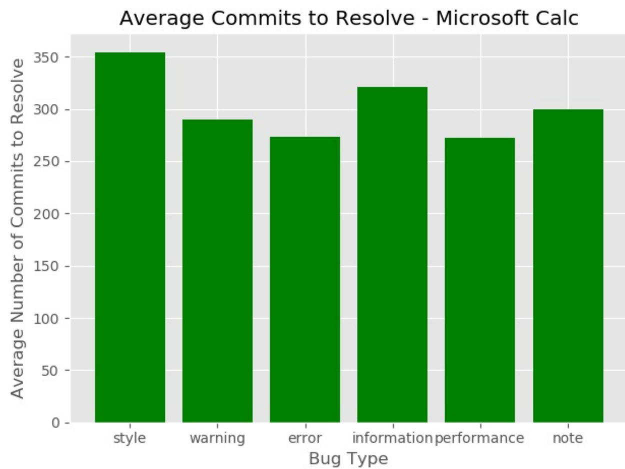

Figure 11. The average number of commits to resolve each type of bug in Microsoft Calculator

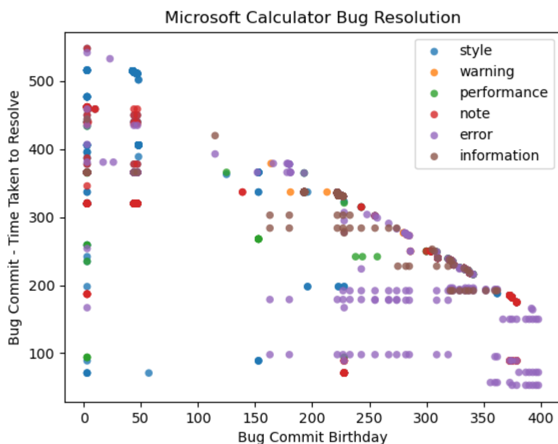

Figure 12. Plotting commit of inception against commits to resolve

4.7 Using Breezy on Microsoft Calculator with Cppcheck

To get a different perspective of Breezy by running on an entirely different code base, we ran Breezy on Microsoft Calculator, a simple C++ based calculator application with only about 400 commits. We also ran this evaluation with Cppcheck instead of Clang. In **Figure 11** we plotted out the average number of commits to resolve each type of bug, and **Figure 12** is a scatter plot that plots the commit the bug was first found on the x-axis with the number of commits it took to resolve on the y-axis. We found that due to the imprecision of Cppcheck bug type parameter, we were unable to come up with in depth analysis on these data points. However, we wish to use these types of graphs to plot other repositories and static analysis tools that we wish to use Breezy with in the future.

4.8 Deploying Breezy on Amazon EC2

To automate running static analysis tools across many instances of a repository, We developed a CLI tool that allows us to create, start, stop, and terminate AWS instances. A simple set-up bash script can be utilized to clone in the repository and all required dependencies. The output of the bug reports can likewise be copied back onto local disk for further data analysis. We used the boto3 [12] library, an Amazon library to manage EC2 instances, and Paramiko [6], a library that can send commands over ssh, in order to complete this CLI.

5 Future Work

5.1 Ranking Bugs

The original motivation for developing the Breezy framework was to find a way to contextually rank the output of static bug checkers by relevance. Static bug checkers are notorious for outputting false positive bugs which makes it less likely for developers to make decisions informed by the vanilla checkers. Some checkers have bug severity as a rank, but those are often hard-coded and do not change in context of the rest of the bugs that a checker outputs.

Breezy was proposed as a tool to generate training data for a model that can rank the output of these checkers. The goal for such a model is to try and predict the amount of time it takes for a given bug to be resolved, in terms of commits. Because Breezy is able to track several bugs in repositories, it also has information about how long it takes for bugs to be resolved. As a result, a model can be trained to take information about the bug such as its code or description and output how long it might take for it to be resolved.

When this is done for many bugs, the bugs can be sorted by their predicted resolve time which may potentially act as a source of bug importance. Even if it is not a good predictor of importance, it can easily filter out bugs that are never resolved by developers, in effect making the ranker a binary filter of whether a bug should be shown to a developer or not. We hope to use Breezy to train such classifiers and potentially built developer tools to filter out 'bad bugs' to make static checkers more readily usable in production environments.

5.2 Large Scale Bug Trends

Although our original goal was to use Breezy as an intermediate step in bug ranking, we now see that it can be used as a much more powerful analytics tool on top of production repositories. consider a production repository that already employs the use of a static bug checker as part of a CI/CD pipeline. Breezy can simply use the output of these test runs to readily generate bug reports without any additional cost. Over time, Breezy is generating bug reports for each commit of the repository, and can readily generate analytics

and trends for the types of bugs that are appearing over time in the repository.

As an example, consider a repository that is migrating from one framework to another and is undergoing a large refactor as a result. Breezy can determine if this is causing a higher than normal influx of buggy code, can draw developers' attention to areas of their codebase where buggy code is manifesting, and hopefully enable developers to prevent making even more serious buggy code changes.

In this sense, Breezy can be used as a novel way of monitoring the health of a production repository and giving developers both fine and coarse grained analysis over the bugs that plague their repositories.

6 Conclusion

We developed Breezy, a novel bug tracking framework that uses existing static analysis tools and a user provided repository to track the repository's bugs over time. We ran Breezy on a couple of test repositories, and the more heavy weight gecko-dev repository, and gained insights into how bugs change over time in repositories. We see Breezy as a stepping stone framework which can be used in a number of ways downstream. Overall, Breezy shows us that looking at bugs at a macro scale and over time can benefit developers greatly.

References

[1] [n.d.]. a C language family frontend for LLVM. ([n. d.]). http://clang.llvm.org/

[2] [n.d.]. Cppcheck. ([n. d.]). https://github.com/danmar/cppcheck

[3] [n.d.]. Infer. ([n. d.]). https://fbinfer.com/

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[5] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[6] Jeff Forcier. 2003. *paramiko*. https://github.com/paramiko/paramiko

[7] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90.

[8] Tim Menzies and Andrian Marcus. 2008. Automated Severity Assessment of Software Defect Reports. *IEEE International Conference on Software Maintenance, ICSM*, 346 – 355. https://doi.org/10.1109/ICSM.2008.4658083

[9] M. Pradel and T. R. Gross. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*. 288–298.

[10] N. Rutar, C. B. Almazan, and J. S. Foster. 2004. A comparison of bug finding tools for Java. In *15th International Symposium on Software Reliability Engineering*. 245–256.

[11] 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 (March 2018), 58–66. https://doi.org/10.1145/3188720

[12] Amazon Web Services. 2014. *boto3*. https://github.com/boto/boto3

[13] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing Bug Finding Tools with Reviews and Tests. In *Testing of Communicating Systems*, Ferhat Khendek and Rachida Dssouli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–55.