

DOI:10.1145/3188720

For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.

BY CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CIERA JASPAN

Lessons from Building Static Analysis Tools at Google

SOFTWARE BUGS COST developers and software companies a great deal of time and money. For example, in 2014, a bug in a widely used SSL implementation (“goto fail”) caused it to accept invalid SSL certificates,³⁶ and a bug related to date formatting caused a large-scale Twitter outage.²³ Such bugs are often statically detectable and are, in fact, obvious upon reading the code or documentation yet still make it into production software.

Previous work has reported on experience applying bug-detection tools to production software.^{6,3,7,29} Although there are many such success stories for developers using static analysis tools, there are also reasons engineers do not always use static analysis tools or ignore their warnings,^{6,7,26,30} including:

Not integrated. The tool is not integrated into the developer’s workflow or takes too long to run;

Not actionable. The warnings are not actionable;

Not trustworthy. Users do not trust the results due to, say, false positives;

Not manifest in practice. The reported bug is theoretically possible, but the problem does not actually manifest in practice;

» key insights

- **Static analysis authors should focus on the developer and listen to their feedback.**
- **Careful developer workflow integration is key for static analysis tool adoption.**
- **Static analysis tools can scale by crowdsourcing analysis development.**

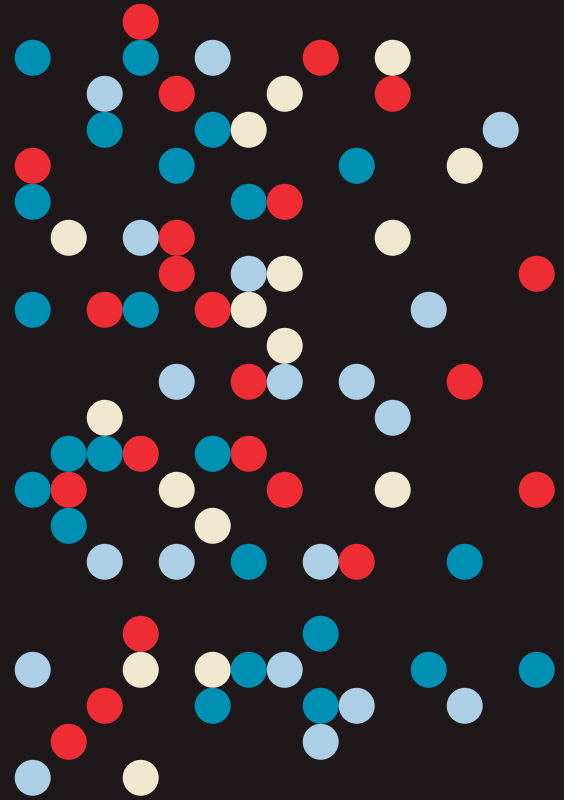
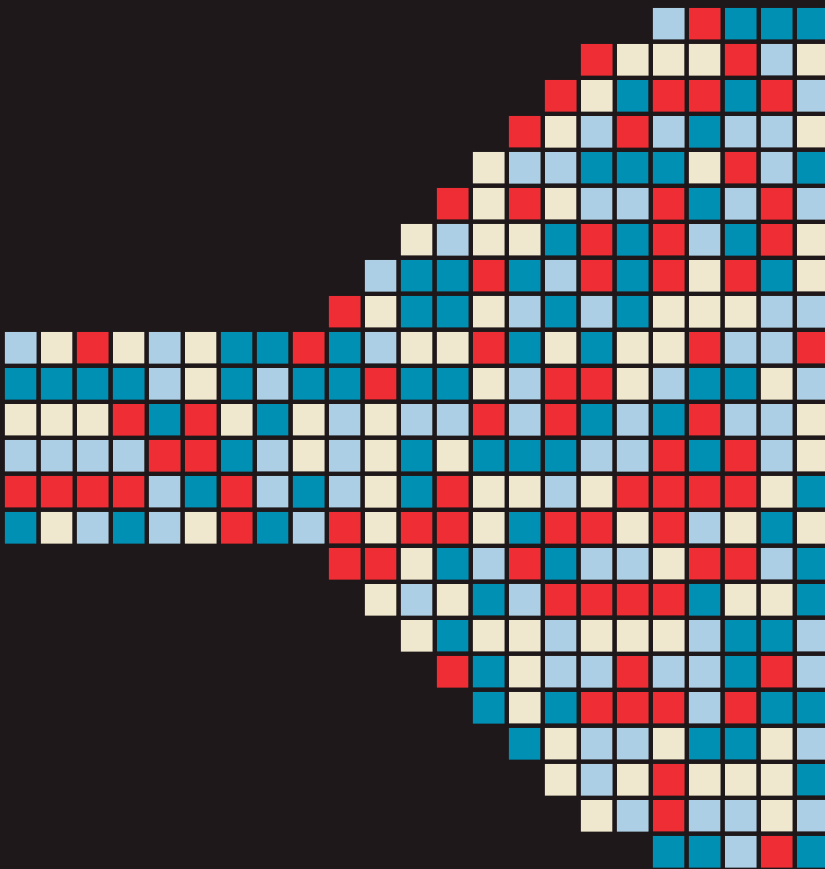


IMAGE BY IGOR KISSELEV

Too expensive to fix. Fixing the detected bug is too expensive or risky; and *Warnings not understood.* Users do not understand the warnings.

Here, we describe how we have applied the lessons from Google's previous experience with FindBugs Java analysis, as well as from the academic literature, to build a successful static analysis infrastructure used daily by most software engineers at Google. Google's tooling detects thousands of problems per day that are fixed by engineers, by their own choice, before the problematic code is checked into Google's companywide codebase.

Scope. We focus on static analysis tools that have become part of the core developer workflow at Google and used by a large fraction of Google's develop-

ers. Many of the static analysis tools deployed at the scale of Google's two-billion-line codebase³² are relatively simple; running more sophisticated analyses at scale is not yet considered a priority.

Note that developers outside of Google working in specialized fields (such as aerospace¹³ and medical devices²¹) may use additional static analysis tools and workflows. Likewise, developers working on specific types of projects (such as kernel code and device drivers⁴) may run ad hoc analyses. There has been lots of great work on static analysis, and we do not claim the lessons we report here are unique, but we do believe that collating and sharing what has worked to improve code quality and the devel-

oper experience at Google is valuable.

Terminology. We use the following terms: analysis tools run one or more "checks" over source code and identify "issues" that may or may not represent actual software faults. We consider an issue to be an "effective false positive" if developers did not take positive action after seeing the issue.³⁵ If an analysis incorrectly reports an issue, but developers make the fix anyway to improve code readability or maintainability, that is not an effective false positive. If an analysis reports an actual fault, but the developer did not understand the fault and therefore took no action, that is an effective false positive. We make this distinction to emphasize the importance of developer perception. Developers, not tool authors, will determine and act on a tool's perceived

false-positive rate.

How Google builds software. Here, we outline key aspects of Google's software-development process. At Google, nearly all developer tools (with the exception of the development environment) are centralized and standardized. Many parts of the infrastructure are built from scratch and owned by internal teams, giving the flexibility to experiment.

Source control and code ownership. Google has developed and uses a single-source control system and a single monolithic source code repository that holds (nearly) all Google proprietary source code.^a Developers use "trunk-based" development, with limited use of branches, typically for releases, not for features. Any engineer can change any piece of code, subject to approval by the code's owners. Code ownership is path-based; an owner of a directory implicitly owns all subdirectories as well.

Build system. All code in Google's repository builds with a customized version of the Bazel build system,⁵ requiring that builds be hermetic; that is, all inputs must be explicitly declared and stored in source control so the builds are easily distributed and parallelized. In Google's build system, Java rules depend on the Java Development Kit and Java compiler that are checked into source control, and such binaries can be updated for all users simply by checking-in new versions. Builds are generally from source (at head), with few binary artifacts checked into the repository. Since all developers use the same build system, it is the source of truth for whether any given piece of code compiles without errors.

Analysis tools. The static analysis tools Google uses are typically not complex. Google does not have infrastructure support to run interprocedural or whole-program analysis at Google scale, nor does it use advanced static analysis techniques (such as separation logic⁷) at scale. Even simple checks have required analysis infrastructure supporting workflow integration to make them successful. The types of analyses deployed as part of the general developer workflow include:

Style checkers (such as Checkstyle,¹⁰

a Google's large open source projects (such as Android and Chrome) use separate infrastructure and their own workflows.

Developers, not tool authors, will determine and act on a tool's perceived false-positive rate.

Pylint,³⁴ and Golint¹⁸);

Bug-finding tools that may extend the compiler (such as Error Prone,¹⁵ ClangTidy,¹² Clang Thread Safety Analysis,¹¹ Govet,¹⁷ and the Checker Framework⁹), including, but not limited to, abstract-syntax-tree pattern-match tools, type-based checks, and unused variable analysis;

Analyzers that make calls to production services (such as to check whether an employee mentioned in a code comment is still employed at Google); and

Analyzers that examine properties of build outputs (such as the size of binaries).

The "goto fail" bug³⁶ would have been caught by Google's C++ linter that checks whether `if` statements are followed by braces. The code that caused the Twitter outage²³ would not compile at Google because of an Error Prone compiler error, a pattern-based check that identifies date-formatting misuses. Google developers also use dynamic analysis tools (such as AddressSanitizer) to find buffer overruns and ThreadSanitizer to find data races.¹⁴ These tools are run during testing and sometimes also with production traffic.

Integrated Development Environments (IDEs). An obvious workflow integration point to show static analysis issues early in the development process is within an IDE. However, Google developers use a wide variety of editors, making it difficult to consistently detect bugs by all developers prior to invoking the build tool. Although Google does use analyses integrated with popular internal IDEs, requiring a particular IDE with analyses enabled is a non-starter.

Testing. Nearly all Google code includes corresponding tests, ranging from unit tests all the way to large-scale integration tests. Tests are integrated as a first-class concept in the build system and hermetic and distributed, just like builds. For most projects, developers write and maintain the tests for their code; projects typically have no separate testing or quality-assurance group. Google's continuous build-and-test system runs tests on every commit and notifies a developer if the developer's change broke the build or caused a test to fail. It also supports testing a change before committing to avoid

breaking downstream projects.

Code review. Every commit to Google's codebase goes through code review first. Although any developer can propose a change to any part of Google's code, an owner of the code must review and approve the change before submission. In addition, even owners must have their code reviewed before committing a change. Code review happens through a centralized, web-based tool that is tightly integrated with other development infrastructure. Static analysis results are surfaced in code review.

Releasing code. Google teams release frequently, with much of the release validation and deployment process automated through a "push on green" methodology,²⁷ meaning an arduous, manual-release-validation process is not possible. If Google engineers find a bug in a production service, a new release can be cut and deployed to production servers at relatively low cost compared with applications that must be shipped to users.

What We Learned from FindBugs

Earlier research, from 2008 to 2010, on static analysis at Google focused on Java analysis with FindBugs^{2,3}: a stand-alone tool created by William Pugh of the University of Maryland and David Hovemeyer of York College of Pennsylvania that analyzes compiled Java class files and identifies patterns of code that lead to bugs. As of January 2018, FindBugs was available at Google only as a command-line tool used by few engineers. A small Google team, called "BugBot," worked with Pugh on three failed attempts to integrate FindBugs into the Google developer workflow.

We have thus learned several lessons:

Attempt 1. Bug dashboard. Initially, in 2006, FindBugs was integrated as a centralized tool that ran nightly over the entire Google codebase, producing a database of findings engineers could examine through a dashboard. Although FindBugs found hundreds of bugs in Google's Java codebase, the dashboard saw little use because a bug dashboard was outside the developers' usual workflow, and distinguishing between new and existing static-analysis issues was distracting.

Attempt 2. Filing bugs. The BugBot team then began to manually triage new issues found by each nightly Find-

Bugs run, filing bug reports for the most important ones. In May 2009, hundreds of Google engineers participated in a companywide "Fixit" week, focusing on addressing FindBugs warnings.³ They reviewed a total of 3,954 such warnings (42% of 9,473 total), but only 16% (640) were actually fixed, despite the fact that 44% of reviewed issues (1,746) resulted in a bug report being filed. Although the Fixit validated that many issues found by FindBugs were actual bugs, a significant fraction were not important enough to fix in practice. Manually triaging issues and filing bug reports is not sustainable at a large scale.

Attempt 3. Code review integration. The BugBot team then implemented a system in which FindBugs automatically ran when a proposed change was sent for review, posting results as comments on the code-review thread, something the code-review team was already doing for style/formatting issues. Google developers could suppress false positives and apply FindBugs' confidence in the result to filter comments. The tooling further attempted to show only new FindBugs warnings but sometimes miscategorized issues as new. Such integration was discontinued when the code-review tool was replaced in 2011 for two main reasons: the presence of effective false positives caused developers to lose confidence in the tool, and developer customization resulted in an inconsistent view of analysis results.

Make It a Compiler Workflow

Concurrent with FindBugs experimentation, the C++ workflow at Google was improving with the addition of new checks to the Clang compiler. The Clang team implemented new compiler checks, along with suggested fixes, then used ClangMR³⁸ to run the updated compiler in a distributed way over the entire Google codebase, refine checks, and programmatically fix all existing instances of a problem in the codebase. Once the codebase was cleansed of an issue, the Clang team enabled the new diagnostic as a compiler error (not a warning, which the Clang team found Google developers ignored) to break the build, a report difficult to disregard. The Clang team was very successful improving the co-

debase through this strategy.

We followed this design and built a simple pattern-based static analysis for Java called Error Prone¹⁵ on top of the javac Java compiler.¹ The first check rolled out, called `PreconditionsCheckNotNull`,^b detects cases in which a runtime precondition check trivially succeeds because the arguments in the method call are transposed, as when, say, `checkNotNull("uid was null", uid)` instead of `checkNotNull(uid, "uid was null")`.

In order to launch checks like `PreconditionsCheckNotNull` without breaking any continuous builds, the Error Prone team runs such checks over the whole codebase using a javac-based MapReduce program, analogous to ClangMR, called JavacFlume built using FlumeJava.⁸ JavacFlume emits a collection of suggested fixes, represented as diffs, that are then applied to produce a whole-codebase change. The Error Prone team uses an internal tool, Rosie,³² to split the large-scale change into small changes that each affect a single project, test those changes, and send them for code review to the appropriate team. The team reviews only those fixes that apply to its code, and, when they approve them, Rosie commits the change. All changes are eventually approved, the existing issues are fixed, and the team enables the compiler error.

When we have surveyed developers who received these patches, 57% of them who received a proposed fix to checked-in code were happy to have received it, and 41% were neutral. Only 2% responded negatively, saying, "It just created busywork for me."

Value of compiler checks. Compiler errors are displayed early in the development process and integrated into the developer workflow. We have found expanding the set of compiler checks to be effective for improving code quality at Google. Because checks in Error Prone are self-contained and written against the javac abstract syntax tree, rather than bytecode (unlike FindBugs), it is relatively easy for developers outside the team to contribute checks. Leveraging these contributions is vital in increasing Error Prone's overall im-

b <http://errorprone.info/bugpattern/PreconditionsCheckNotNull>

pact. As of January 2018, 733 checks had been contributed by 162 authors.

Reporting issues sooner is better. Google's centralized build system logs all builds and build results, so we identified all users who had seen one of the error messages in a given time window. We sent a survey to developers who recently encountered a compiler error and developers who had received a patch with a fix for the same problem. Google developers perceive that issues flagged at compile time (as opposed to patches for checked-in code) catch more important bugs; for example, survey participants deemed 74% of the issues flagged at compile time as "real problems," compared to 21% of those found in checked-in code. In addition, survey participants deemed 6% of the issues found at compiletime (vs. 0% in checked-in code) "critical." This result is explained by the "survivor effect";³ that is, by the time code is submitted, the errors are likely to have been caught by more expensive means (such as testing and code review). Moving as many checks into the compiler as possible is one proven way to avoid those costs.

Criteria for compiler checks. To scale-up our work, we have defined criteria for enabling checks in the compiler, setting the bar high, since breaking the compile would be a significant disruption. A compiler check at Google should be easily understood; actionable and easy to fix (whenever possible, the error should include a suggested fix that can be applied mechanically); produce no effective false positives (the analysis should never stop the build for correct code); and report issues affecting only correctness rather than style or best practices.

The primary goal of an analyzer satisfying these criteria is not simply to detect faults but to automatically fix all instances of a prospective compiler error throughout the codebase. However, such criteria limit the scope of the checks the Error Prone team enables when compiling code; many issues that cannot always be detected correctly or mechanically fixed are still serious problems.

Warn During Code Review

Once the Error Prone team had built the infrastructure needed to detect issues at compile time, and had proved

the approach works, we wanted to show more high-impact bugs that do not meet the criteria we outlined earlier for compiler errors and provide results for languages other than Java and C++. The second integration point for static analysis results is Google's code review tool, Critique; static analysis results are exposed in Critique using Tricorder,³⁵ Google's program-analysis platform. As of January 2018, there was a compiler warnings-free default for C++ and Java builds at Google, with all analysis results either shown as compiler errors or in code review.

Criteria for code-review checks. Unlike compile-time checks, analysis results shown during code review are allowed to include up to 10% effective false positives. There is an expectation during code review that feedback is not always perfect and that authors evaluate proposed changes before applying them. A code review check at Google should fulfill several criteria:

Be understandable. Be easy for any engineer to understand;

Be actionable and easy to fix. The fix may require more time, thought, or effort than a compiler check, and the result should include guidance as to how the issue might indeed be fixed;

Produce less than 10% effective false positives. Developers should feel the check is pointing out an actual issue at least 90% of the time;^c and

Have the potential for significant impact on code quality. The issues may not affect correctness, but developers should take them seriously and deliberately choose to fix them.

Some issues are severe enough to be flagged in the compiler, but producing them or developing an automated fix is not feasible. For example, fixing an issue may require significant restructuring of the code. Enabling these checks as compiler errors would require manual cleanup of existing instances that is infeasible on the scale of Google's vast codebase. Analysis tools show these checks in code review prevent new occurrences of the issue, allowing the developer to decide how to

make an appropriate fix. Code review is also a good context for reporting relatively less-important issues like stylistic problems or opportunities to simplify code. In our experience, reporting them at compile-time is frustrating for developers and makes it more difficult to iterate and debug quickly; for example, an unreachable code detector might hinder attempts to temporarily disable a block of code for debugging. However, at code-review time, developers are preparing their code to be seen; they are already in a critical mindset and more receptive to seeing readability and stylistic details.

Tricorder. Tricorder is designed to be easily extensible and support many different kinds of program-analysis tools, including static and dynamic analyses. We showed a suite of Error Prone checks in Tricorder that cannot be enabled as compiler errors. Error Prone also inspired a new set of analyses for C++ that are integrated with Tricorder and called ClangTidy.¹² Tricorder analyzers report results for more than 30 languages, support simple syntactic analyses like style checkers, leverage compiler information for Java, JavaScript, and C++, and are straightforward to integrate with production data (such as about jobs that are currently running). Tricorder continues to be successful at Google because it is a plug-in model supporting an ecosystem of analysis writers, actionable issues are highlighted during the code-review process, and it provides feedback channels to improve analyzers and ensure analyzer developers act on the feedback.

Empower users to contribute. As of January 2018, Tricorder included 146 analyzers, with 125 contributed from outside the Tricorder team and seven plug-in systems for hundreds of additional checks (such as ErrorProne and ClangTidy, which comprise two of the seven analyzers plug-in systems).

Provide fixes and involve reviewers. Tricorder checks can provide suggested fixes that can be directly applied from the code-review tool. They are seen by both the reviewer and the author, and the reviewer can ask the author to fix the problematic code simply by clicking a "Please fix" button on the analysis result. Reviewers typically withhold approval of a change until

^c Although this number was initially chosen by the first author somewhat arbitrarily, it seems to be a sweet spot for developer satisfaction and matches the cutoff for similar systems in other companies.


all their comments, manual and automated, have been addressed.

Iterate on feedback from users. In addition to the “Please fix” button, Tricorder also provides a “Not useful” button that reviewers or proposers can click to express that they do not like the analysis finding. Clicking automatically files a bug in the issue tracker, routing it to the team that owns the analyzer. The Tricorder team tracks such not-useful clicks, computing the ratio of “Please fix” vs. “Not useful” clicks. If the ratio for an analyzer goes above 10%, the Tricorder team disables the analyzer until the author(s) improve it. While the Tricorder team has rarely had to permanently disable an analyzer, it has disabled an analyzer (on several occasions) while the analyzer author is removing and revising sub-checks that were particularly noisy.


The bugs being filed often lead to improvement in the analyzers that in turn greatly improves developers’ satisfaction with those analyzers; for example, the Error Prone team developed, in 2014, an Error Prone check that flagged when too many arguments were being passed to a `printf`-like function in Guava.¹⁹ The `printf`-like function did not actually accept all `printf` specifiers, accepting only `%s`. About once per week the Error Prone team would receive a “Not useful” bug claiming the analysis was incorrect because the number of format specifiers in the bug filers’ code matched the number of arguments passed. In every case, the analysis was correct, and the user was trying to pass specifiers other than `%s`. The team thus changed the diagnostic text to state directly that the function accepts only the `%s` placeholder and stopped getting bugs filed about that check.

Scale of Tricorder. As of January 2018, Tricorder had analyzed approximately 50,000 code review changes per day. During peak hours, there were three analysis runs per second. Reviewers clicked “Please Fix” more than 5,000 times per day, and authors applied the automated fixes approximately 3,000 times per day. And Tricorder analyzers received “Not useful” clicks 250 times per day.

The success of code-review analysis suggests it occupies a “sweet spot” in the developer workflow at Google.



Even in a mature codebase with full test coverage and a rigorous code-review process, bugs slip by.



Analysis results that are shown at compilation time must reach a much higher bar for quality and accuracy that is not possible to meet for some analyses that can still identify serious faults. After the review and code are checked in, the friction confronting developers for making changes increases. Developers are thus hesitant to make additional changes to code that has already been tested and released, and lower severity and less-important issues are unlikely to be addressed. Other analysis projects among major software-development organizations (such as Facebook Infer analysis for Android/iOS apps⁷) have also highlighted code review as a key point for reporting analysis results.

Expand Analyzer Reach

As Google developer-users have gained trust in the results from Tricorder analyzers, they continue to request further analyses. Tricorder addresses this in two ways: allowing project-level customization and adding analysis results at additional points in the developer workflow. In this section, we also touch on the reasons Google does not yet leverage more sophisticated analysis techniques as part of its core developer workflow.

Project-level customization. Not all requested analyzers are equally valuable throughout the Google codebase; for example, some analyzers are associated with higher false-positive rates and so would have correspondingly high effective false-positive rates or require specific project configuration to be useful. These analyzers all have value but only for the right team.

To satisfy these requests, we aimed to make Tricorder customizable. Our previous experience with customization for FindBugs did not end well; user-specific customization caused discrepancies within and across teams and resulted in declining use of tools. Because each user could see a different view of issues, there was no way to ensure a particular issue was seen by everyone working on a project. If developers removed all unused imports from their team’s code, the fix would quickly backslide if even a single other developer was not consistent about removing unused imports.

To avoid such problems, Tricorder allows configuration only at the proj-

ect level, ensuring that anyone making a change to a particular project sees a consistent view of the analysis results relevant to that project. Maintaining a consistent view has enabled several types of analyzers to do the following:

Produce dichotomous results. For example, Tricorder includes an analyzer for protocol buffer definitions³³ that identifies changes that are not backward compatible. It is used by developer teams that ensure persistent information from protocol buffers in their serialized form but is annoying for teams that do not store data in this form. Another example is an analyzer that suggests using Guava³⁷ or Java 7 idioms that do not make sense for projects that cannot use these libraries or language features;


Need a particular setup or in-code annotations. For example, teams can only use the Checker Framework's nullness analysis⁹ if their code is annotated appropriately. Another analysis, when configured, will check the increase in binary size and method count for a particular Android binary and warn developers if there is a significant increase or if they are approaching a hard limit;

Support custom domain-specific languages (DSLs) and team-specific coding guidelines. Some Google software development teams have developed small DSLs with associated validators they wish to run. Other teams have developed their own best practices for readability and maintainability and would like to enforce those checks; and


Are highly resource-intensive. An example is hybrid analyses that incorporate results from dynamic analysis. Such analyses provide high value for some teams but are too costly or slow for all.

As of January 2018, there were approximately 70 optional analyses available within Google, and 2,500 projects had enabled at least one of them. Dozens of teams across the company are actively developing a new analyzer, most outside the developer-tools group.

Additional workflow integration points. As developers have gained trust in the tools, they have also requested further integration into their workflow. Tricorder now provides analysis results through a command-line tool, a continuous integration system, and a code-browsing tool.



Engineers working on static analysis must demonstrate impact through hard data.



Command line support. The Tricorder team added command-line support for developers who are, in effect, code janitors, regularly going through and scrubbing their team's codebase of various analysis warnings. These developers are also very familiar with the types of fixes each analysis will generate and have high trust in specific analyzers. Developers can thus use a command-line tool to automatically apply all fixes from a given analysis and generate cleanup changes;

Gating commits. Some teams want specific analyzers to actually block commits, rather than just appear in the code-review tool. The ability to block commits is commonly requested by teams that have highly specific custom checks with no false positives, usually for a custom DSL or library; and

Results in code browsing. Code browsing works best for showing the scale of a problem across a large project (or an entire codebase). For example, analysis results when browsing code about a deprecated API can show how much work a migration entails; or some security and privacy analyses are global in scope and require specialized teams to vet the results before determining whether there is indeed a problem. Since analysis results are not displayed by default, the code browser allows specific teams to enable an analysis layer and then scan the entire codebase and vet the results without disrupting other developers with distractions from these analyzers. If an analysis result has an associated fix, then developers can apply the fix with a single click from the code-browsing tool. The code browser is also ideal for displaying results from analyses that utilize production data, as this data is not available until code is committed and running.

Sophisticated analyses. All of the static analyses deployed widely at Google are relatively simple, although some teams work on project-specific analysis frameworks for limited domains (such as Android apps) that do interprocedural analysis. Interprocedural analysis at Google scale is technically feasible. However, implementing such an analysis is very challenging. All of Google's code resides in a single monolithic source code repository, as discussed, so, conceptually, any code in the repository can be part of any binary. It is thus possible to imagine

a scenario in which analysis results for a particular code review would require analyzing the entire repository. Although Facebook's Infer^{7,25} focuses on compositional analysis in order to scale separation-logic-based analysis to multimillion-line repositories, scaling such analysis to Google's multibillion-line repository would still take significant engineering effort.

As of January 2018, implementing a system to do more sophisticated analyses has not been a priority for Google since:

Large investment. The up-front infrastructure investment would be prohibitive;

Work needed to reduce false-positive rates. Analysis teams would have to develop techniques to dramatically reduce false-positive rates for many research analyzers and/or severely restrict which errors are displayed, as with Infer;

Still more to implement. Analysis teams still have plenty more "simple" analyzers to implement and integrate; and

High upfront cost. We have found the utility of such "simple" analyzers to be high, a core motivation of FindBugs.²⁴ In contrast, even determining the cost-benefit ratio for more complicated checks has a high up-front cost.

Note this cost-benefit analysis may be very different for developers outside of Google working in specialized fields (such as aerospace¹³ and medical devices²¹) or on specific projects (such as device drivers⁴ and phone apps⁷).

Lessons

Our experience attempting to integrate static analysis into Google's workflow taught us valuable lessons:

Finding bugs is easy. When a codebase is large enough, it will contain practically any imaginable code pattern. Even in a mature codebase with full test coverage and a rigorous code-review process, bugs slip by. Sometimes the problem is not obvious from local inspection, and sometimes bugs are introduced by seemingly harmless refactorings. For example, consider the following code snippet hashing a field `f` of type `long`

```
result =
    31 * result
    + (int) (f ^ (f >>> 32));
```

Now consider what happens if the developer changes the type of `f` to `int`. The code continues to compile, but the right shift by 32 becomes a no-op, the field is XORed with itself, and the hash for the field becomes a constant 0. The result is `f` no longer affects the value produced by the `hashCode` method. The right shift by more than 31 is statically detectable by any tool able to compute the type of `f`, yet we fixed 31 occurrences of this bug in Google's codebase while enabling the check as a compiler error in Error Prone.

Since finding bugs is easy,²⁴ Google uses simple tooling to detect bug patterns. Analysis writers then tune the checks based on results from running over Google code.

Most developers will not go out of their way to use static analysis tools. Following in the footsteps of many commercial tools, Google's initial implementation of FindBugs relied on engineers choosing to visit a central dashboard to see the issues found in their projects, though few of them actually made such a visit. Finding bugs in checked-in code (that may already be deployed and running without user-visible problems) is too late. To ensure that most or all engineers see static-analysis warnings, analysis tools must be integrated into the workflow and enabled by default for everyone. Instead of providing bug dashboards, projects like Error Prone extend the compiler with additional checks, and surface analysis results in code review.

Developer happiness is key. In our experience and in the literature, many attempts to integrate static analysis into a software-development organization fail. At Google, there is typically no mandate from management that engineers use static analysis tools. Engineers working on static analysis must demonstrate impact through hard data. For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.

To build a successful analysis platform, we have built tools that deliver high value for developers. The Tricorder team keeps careful accounting of issues fixed, performs surveys to understand developer sentiment, makes it easy to file bugs against the analysis tools, and uses all this data to justify continued investment. Developers need to build trust in analysis tools. If

a tool wastes developer time with false positives and low-priority issues, developers will lose faith and ignore results.

Do not just find bugs, fix them. To sell a static analysis tool, a typical approach is to enumerate a significant number of issues that are present in a codebase. The intent is to influence decision makers by indicating a potential ability to correct the underlying bugs or prevent them in the future. However, that potential will remain unrealized if developers are not incentivized to act. This is a fundamental flaw: analysis tools measure their utility by the number of issues they identify, while integration attempts fail due to the low number of bugs actually fixed or prevented. Instead, Google static analysis teams take responsibility for fixing, as well as finding, bugs, and measure success accordingly. Focusing on fixing bugs has ensured that tools provide actionable advice³⁰ and minimize false positives. In many cases, fixing bugs is as easy as finding them through automated tooling. Even for difficult-to-fix issues, research over the past five years has highlighted new techniques for automatically creating fixes for static analysis issues.^{22,28,31}

Crowdsource analysis development. Although typical static analysis tools require expert developers to write the analyses, experts may be scarce and not actually know what checks will have the greatest impact. Moreover, analysis experts are typically not domain experts (such as those working with APIs, languages, and security). With FindBugs integration, only a small number of Googlers understood how to write new checks, so the small BugBot team had to do all the work themselves. This limited the velocity of adding new checks and prevented others from contributing their domain knowledge. Teams like Tricorder now focus on lowering the bar to developer-contributed checks, without requiring prior static analysis experience. For example, the Google tool Refaster³⁷ allows developers to write checks by specifying example before and after code snippets. Since contributors are frequently motivated to contribute after debugging faulty code themselves, new checks are biased toward those that save developer time.

Conclusion

Our most important insight is that careful developer workflow integration is key for static analysis tool adoption. While tool authors may believe developers should be delighted by a list of probable defects in code they have written, in practice we did not find such a list motivates developers to fix the defects. As analysis-tool developers, we must measure our success in terms of defects corrected, not the number presented to developers. This means our responsibility extends far beyond the analysis tool itself.

We advocate for a system focused on pushing workflow integration as early as possible. When possible, checks are enabled as compiler errors. To avoid breaking builds, tool writers take on the task of first fixing all the existing issues in the codebase, allowing us to “ratchet” the quality of Google’s codebase one small step at a time, without regressions. Since we present the errors in the compiler, developers encounter them immediately after writing code, while they are still amenable to making changes. To enable this, we have developed infrastructure for running analyses and producing fixes over the whole vast Google codebase. We also benefit from code review and submission automation that allows a change to hundreds of files, as well as an engineering culture in which changes to legacy code are typically approved because improving the code wins over risk aversion.

Code review is a sweet spot for displaying analysis warnings before code is committed. In order to ensure developers are receptive to analysis results, Tricorder presents issues only when a developer is changing the code in question, before the change is committed, and the Tricorder team applies a set of criteria to selecting what warnings to display. Tricorder further gathers user data in the code-review tool that is used to detect any analyses that produce unacceptable numbers of negative reactions. The Tricorder team minimizes effective false positives by disabling misbehaving analyses.

To overcome warning blindness, we have worked to regain the trust of Google engineers, finding Google developers have a strong bias to ignore static analysis, and any false positives

or poor reporting give them a justification for inaction. Analysis teams are quite careful to enable a check as an error or warning only after vetting it against the criteria described here, so developers are rarely inundated, confused, or annoyed by analysis results. Surveys and feedback channels are an important quality control for this process. Now that developers have gained trust in analysis results, the Tricorder team is fulfilling requests for more analyses surfaced in more locations in the Google developer workflow.

We have built a successful static analysis infrastructure at Google that prevents hundreds of bugs per day from entering the Google codebase, both at compiletime and during code reviews. We hope others can benefit from our experience to successfully integrate static analyses into their own workflows. C

References

1. Aftandilian, E., Sauciu, R., Priya, S., and Krishnan, S. Building useful program analysis tools using an extensible compiler. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation* (Riva del Garda, Italy, Sept. 23–24). IEEE Computer Society Press, 2012, 14–23.
2. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., and Pugh, W. Using static analysis to find bugs. *IEEE Software* 25, 5 (Sept.–Oct. 2008), 22–29.
3. Ayewah, N. and Pugh, W. The Google FindBugs fixit. In *Proceedings of the International Symposium on Software Testing and Analysis* (Trento, Italy, July 12–16). ACM Press, New York, 2010.
4. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., and Ustuner, A. Thorough static analysis of device drivers *ACM SIGOPS Operating Systems Review* 40, 4 (Oct. 2006), 73–85.
5. Bazel; <http://www.bazel.io>
6. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later. *Commun. ACM* 53, 2 (Feb. 2010), 66–75.
7. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving fast with software verification. In *Proceedings of the NASA Formal Method Symposium* (Pasadena, CA, Apr. 27–29). Springer, 2015.
8. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., and Weizenbaum, N. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Canada, June 5–10). ACM Press, New York, 2010.
9. The Checker Framework; <https://checkerframework.org>
10. Checkstyle Java Linter; <http://checkstyle.sourceforge.net/>
11. Clang Thread Safety Analysis; <http://clang.lvm.org/docs/ThreadSafetyAnalysis.html>
12. ClangTidy; <http://clang.lvm.org/extra/clang-tidy.html>
13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTREE analyzer. In *Proceedings of the European Symposium on Programming* (Edinburgh, Scotland, Apr. 2–10). Springer, Berlin, Heidelberg, 2005.
14. Dynamic Sanitizer Tools; <https://github.com/google/sanitizers>
15. Error Prone; <http://errorprone.info>
16. FindBugs; <http://findbugs.sourceforge.net/>
17. Go vet; <https://golang.org/cmd/vet>
18. Golint; <https://github.com/golang/lint>

19. Grammatech; <https://resources.grammatech.com/medical>
20. Griesmayer, A., Bloem, R., Cook, B. Repair of Boolean programs with an application to C. In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA, Aug. 17–20). Springer, Berlin, New York, 2006.
21. Guava: Google Core Libraries for Java 1.6+; <https://code.google.com/p/guava-libraries/>
22. Gupta, P., Ivey, M., and Penix, J. Testing at the speed and scale of Google. *Google Engineering Tools Blog*, 2011; <http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html>
23. Hacker News. Twitter outage report, 2016; <https://news.ycombinator.com/item?id=8810157>
24. Hovemeyer, D. and Pugh, W. Finding bugs is easy. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 92–106.
25. Infer; <http://fbinfer.com/>
26. Johnson, B., Song, Y., Murphy-Hill, E.R., and Bowdidge, R.W. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 18–26). ACM Press, New York, 2013.
27. Klein, D.V., Betser, D.M., and Monroe, M.G. Making ‘push on green’ a reality: Issues and actions involved in maintaining a production service. *Jagim*, 39, 5 (2014), 26–32.
28. Kneuss, E., Koukoutos, M., and Kuncak, V. Deductive program repair. In *Proceedings of the 27th International Conference on Computer Aided Verification* (San Francisco, CA, July 18–24). Springer, 2015.
29. Larus, J.R., Ball, T., Das, M., DeLine, R., Fahndrich, M., Pincus, J., Rajamani, S.K., and Venkatapathy, R. Righting software. *IEEE Software* 21, 3 (May 2004), 92–100.
30. Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Whitehead, Jr., E. J. Does bug prediction support human developers’ findings?: From a Google case study. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 18–26). ACM Press, New York, 2013.
31. Logozzo, F. and Ball, T. Modular and verified automatic program repair. *ACM SIGPLAN Notices* 46, 10 (Oct. 19, 2012), 133–146.
32. Potvin, R. and Levenberg, J. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (July 2016), 78–87.
33. Protocol buffers; <http://code.google.com/p/protobuf/>
34. Pylint Python Linter; <http://www.pylint.org/>
35. Sadowski, C., van Gogh, J., Jaspán, C., Söderberg, E., and Winter, C. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering* (Firenze, Italy, May 16–24). ACM Press, New York, 2015.
36. Synopsys Editorial Team. *Coverity Report on the ‘Goto Fail’ Bug*. Blog post, Synopsys, Mountain View, CA, Feb. 25, 2014; <http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-goto-fail.html>
37. Wasserman, L. Scalable, example-based refactorings with Refaster. In *Proceedings of the Workshop on Refactoring Tools* (Indianapolis, IN, Oct. 26). ACM Press, New York, 2013.
38. Wright, H., Jasper, D., Klimek, M., Carruth, C., and Wan, Z. Large-scale automated refactoring using ClangMR. In *Proceedings of the 29th IEEE International Conference on Software Maintenance* (Eindhoven, the Netherlands, Sept. 22–28). IEEE Computer Society Press, 2013.

Caitlin Sadowski (supertri@google.com) is a software engineer at Google Inc., Mountain View, CA, USA.

Edward Aftandilian (eaftan@google.com) leads the Java compiler and static analysis team at Google, Inc., Mountain View, CA, USA.

Alex Eagle (alexeagle@google.com) is a software engineer at Google Inc., Mountain View, CA, USA.

Liam Miller-Cushon (cushon@google.com) is a software engineer at Google Inc., Mountain View, CA, USA.

Ciera Jaspán (ciera@google.com) is a software engineer at Google Inc., Mountain View, CA, USA.

Copyright held by the authors.
Publication rights licensed to ACM. \$15.00