2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Tricorder: Building a Program Analysis Ecosystem

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, Collin Winter
{supertri, jvg, ciera, emso, collinwinter}@google.com
Google, Inc.

***Abstract*—Static analysis tools help developers find bugs, improve code readability, and ensure consistent style across a project. However, these tools can be difficult to smoothly integrate with each other and into the developer workflow, particularly when scaling to large codebases. We present TRICORDER, a program analysis platform aimed at building a data-driven ecosystem around program analysis. We present a set of guiding principles for our program analysis tools and a scalable architecture for an analysis platform implementing these principles. We include an empirical, *in-situ* evaluation of the tool as it is used by developers across Google that shows the usefulness and impact of the platform.**

***Index Terms*—program analysis, static analysis**

I. INTRODUCTION

Static analysis tools provide a promising way to find bugs in programs before they occur in production systems. Developers can run analyzers on their source code to find issues, before even checking in the code. In spite of the rich vein of research on static analysis tools, [3], [14], [16], these tools are often not used effectively in practice. High false positive rates, confusing output, and poor integration into the developers' workflow all contribute to the lack of use in everyday development activities [23], [27].

In addition to finding bugs, tools must take into account the high demands on a developer's time [26]. Any interruption generated by an automated tool forces a developer to context-switch away from her primary objective [27]. Successful static analysis tools add high value while minimizing the distractions for already-busy software engineers.

Our past experience with static analysis tools also showed that many of them are not scalable to a codebase of Google's size. Analyses cannot presume that they have access to the entire source repository or all the compilation results; there is simply too much data for a single machine. Therefore, analyses *must* be shardable and able to run as part of a distributed computation with only partial information. The sharded analysis must be extremely fast and provide results within a couple of minutes. In our previous experiences at Google, no existing analysis platform could scale in this way.

We also found that existing platforms and tools were not extensible enough. Google has many specialized frameworks and languages, and an ideal system would provide static analyses for all of them. Our ideal system would let domain experts write their own analyses, without having to bear the cost of building or maintaining an entire end-to-end pipeline. For example, a team writing C++ libraries can write checks to make sure developers use those libraries correctly, without worrying about the problems inherent in running a large production system.

After experimenting with a variety of commercial and open-source program analysis tools at Google (see Section II for more details), we repeatedly had problems with tool scalability or usability. Building off of these experiences, we wanted to create a static analysis platform that would:

- be widely and actively used by developers to fix problems in their code, without prompting from a small group of advocates or management.
- integrate smoothly into the existing developer workflow.
- scale to the size of an industrial codebase.
- empower developers, even non-analysis experts, to write and deploy their own static analyses.

In this paper, we present TRICORDER, a program analysis platform aimed at building a data-driven ecosystem around static analysis. TRICORDER integrates static analysis into the workflow of developers at Google, provides a feedback loop between developers and analyzer writers, and simplifies fixing issues discovered by analysis tools. To accomplish this, TRICORDER leverages a microservices architecture to scale to Google's codebase, and generates some 93,000 analysis results each day. A small team of 2-3 people maintain TRICORDER and the ecosystem around it, in addition to working on an open-source version of the platform [35]. TRICORDER's plug-in model allows contributors from teams across the company to become part of the program analysis community at Google.

The contributions of this paper include:

- A set of guiding principles that have resulted in a successful, widely used program analysis platform at Google. (Section III)
- A scalable architecture for a program analysis platform. This platform builds a program analysis ecosystem through workflow integration, supporting contributors, responding to feedback, and automatic fixes. (Section IV)
- An empirical, *in situ* validation of the usefulness of the platform based upon developers' responses to the analyses in their normal workflow. (Section V)

II. BACKGROUND

A. Development workflow

At Google, most engineers[1] work in an extremely large codebase, where most software development occurs at head. Every workday at Google, engineers perform more than 800k

[1]Some exceptions to this development setup exist; for example, Chrome and Android have independent open source repositories and developer tools.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.76

builds, run 100M test cases, produce 2PB of build outputs, and send 30k changelist snapshots (patch diffs) for review.

A large codebase has many benefits, including ease of code reuse and the ability to do large-scale refactorings atomically. Because code reuse is common, most code depends upon a core set of libraries, and making changes to these base libraries may impact many projects. To ensure that changes do not break other projects, Google has a strong testing culture backed by continuous testing infrastructure [43], [15].

Google engineers use a standardized, distributed build system to produce hermetic builds from source code [20]. A team of dedicated engineers maintains this infrastructure centrally, providing a common location to insert analysis tools. Because Google engineers use the same distributed build environment, they may use their choice of editor. Editor choices include, but are not limited to, Eclipse, IntelliJ, emacs, and vim [38].

As part of a strong code review culture, every new patch, called a *changelist*, is reviewed by someone other than the author before being checked in. Engineers perform these reviews using an internal code review tool, similar to Gerrit [19]. This tool provides the ability to comment on lines of code, reply to existing comments, upload new *snapshots* of the code being reviewed (author), and approve the changelist (reviewer).

Instead of using a separate QA process, busy engineers test (and analyze!) their own code. This means that analysis results must target engineers, and it must be easy for those engineers to run and respond to analyzers. Because most code being shipped is server code, the cost of pushing a new version is very low, making it relatively easy to fix bugs after code has shipped.

B. Program analysis at Google

Several attempts have been made to integrate program analysis tools into the Google development workflow. FindBugs [18] in particular has a long history of experimentation at Google [5], [3], [4], along with other analysis tools such as Coverity [12], Klocwork [24], and fault prediction [28]. All of these tools have largely fallen out of use due to problems with workflow integration, scaling, and false positives. Some tools displayed results too late, making developers less likely to fix problems after they had submitted their code. Others displayed results too early, while developers were still experimenting with their code in the editor. Editor-based tools also hit scaling problems: their latency requirements for interactive use were not able to keep up with the size of the codebase. Nearly all tools had to be run as a distinct step, and were difficult to integrate with the standard compiler toolchains. We have repeatedly found that when developers have to navigate to a dashboard or run a standalone command line tool, analysis usage drops off.

Even when developers ran these tools, they often produced high false positive rates and inactionable results [28]. These experiences match prior research showing why developers do not use static analysis tools [23]. In the end, very few developers used any of the tools we previously experimented with, and even for the analysis that got the most traction, FindBugs, the command-line tool was used by only 35 developers in 2014 (and by 20 of those only once). We did previously show FindBugs results in code review [3], but this attempt ran into scaling problems (resulting in stale or delayed results) and produced many results developers were uninterested in addressing. In contrast, TRICORDER has seen success as a part of the standard developer workflow.

III. GOOGLE PHILOSOPHY ON PROGRAM ANALYSIS

A. No false positives

"No" may be a bit of an overstatement, but we severely limit the number of false positives we allow analyses to produce. False positives are bad for both usability and adoption [8], [23], [33].

There is a disconnect around what exactly the term "false positive" means. To an analysis writer, a false positive is an incorrect report produced by their analysis tool. However, to a developer, a false positive is any report that they did not want to see [5].

We prefer to use the term *effective false positive* to capture the developer's perspective. We define an effective false positive as any report from the tool where a user chooses not to take action to resolve the report. As an example of this, some Google developers use static annotation checking systems (e.g. for data race detection [34]). When an annotation checking tool correctly reports an issue, it could mean that either there is a bug in the source code (e.g. a variable is not actually protected by a lock), or that the code is actually fine but the set of annotations is not exhaustive enough for the tool. Typically, in program analysis research, the latter is not considered a false positive – the developer needs to supply additional information to the tool. However, some developers consider such issues to be "false positives" since they do not represent a bug in the code [36].

In contrast, we have found that if an analysis incorrectly reports a bug, but making the suggested fix would improve code readability, this is not considered a false positive. Readability and documentation analyses are frequently accepted by developers, especially if they come with a suggested improvement.

It is also notworthy that some analyses may have false positives in theory, but not in practice. For example, an analysis may have false positives only when a program is constructed in an unusual way, but in practice, such a program is never seen. Such an analysis might have theoretical false positives, but in an environment with a strictly-enforced style guide, it would effectively have zero false positives.

The bottom line is that **developers will decide whether an analysis tool has high impact, and what a false positive is.**

B. Empower users to contribute

In a company using a diverse set of languages and custom APIs, no single team has the domain knowledge to write all needed analyses. Relevant expertise and motivation exists among developers throughout the company, and we want to leverage this existing knowledge by empowering developers to contribute their own analyses. Developer contributions both

enrich the set of available analyses and make users more responsive to analysis results.

However, while these contributors are experts in their domains, they may not have the knowledge, or skill set, to effectively integrate their analyses into the developer workflow. Ideally, workflow integration and the boilerplate needed to get an analysis up and running should not be the concern of the analysis writer. This is where TRICORDER comes in, as a pluggable program analysis platform supporting analysis contributors throughout the company.

In order to keep the quality of analyzers high, we have a "contract" with analyzer writers about when we may remove their analyzer from TRICORDER. That is, we reserve the right to disable analyzers if:

- No one is fixing bugs filed against the analyzer.
- Resource usage (e.g. CPU/disk/memory) is affecting TRICORDER performance. In this case, the analyzer writer needs to start maintaining a standalone service that TRICORDER calls out to (see Section IV-A for more details).
- The analyzer results are annoying developers (see Section IV-E for how we calculate this).

Our experience is that pride-in-work combined with the threat of disabling an analyzer makes the authors highly motivated to fix problems in their analyzers.

C. Make data-driven usability improvements

Responding to feedback is important. Developers build trust with analysis tools, and this trust is quickly lost if they do not understand the tool's output [8]. We also have found (by examining bug reports filed against analyzers) that many analysis results have confusingly worded messages; this is typically an easy problem to fix. For instance, for one analyzer 75% of all bugs filed against the tool from TRICORDER were due to misinterpretations of the result wording and were fixed by updating the message text and/or linking to additional documentation. Establishing a feedback loop to improve the usability of analysis results significantly increases the utility of analysis tools.

D. Workflow integration is key

Integration into developer workflow is a key aspect in making program analysis tools effective [23]. If an analysis tool is a standalone binary that developers are expected to run, it just will not be run as frequently as intended. We posit that analyses should be automatically triggered by developer events such as editing code, running a build, creating/updating a changelist, or submitting a changelist. Analysis results should be shown before code is checked in, because the tradeoffs are different when an engineer has to modify (potentially working) submitted code. As one example of this, we surveyed developers when sending them changelists to fix an error in their code we were planning to turn on as a compiler error, and also when they encountered those errors as compiler errors. Developers were twice as likely to say the error represented a significant bug when encountered as a compiler error. The importance of showing results early matches previous experience with FindBugs [3].

When possible, we integrate static analysis into the build [2]. We support a variety of analyses built on top of the ErrorProne javac extension [17] and the Clang compiler [10]. These analyses break the build when they find an issue, so the effective false positive rate must be essentially zero. They also cannot significantly slow down compiles, so must have < 5% overhead. Ideally, we only show results that cause builds to fail, as we have found warnings shown when building are often ignored. However, build integration is not always practical, e.g. when the false positive rate is too high, the analysis is too time consuming, or it is only worthwhile to show results on newly edited lines of code.

TRICORDER introduces an effective place to show warnings. Given that all developers at Google use code review tools before submitting changes, TRICORDER's primary use is to provide analysis results at code review time. This has the added benefit of enabling peer accountability, where the reviewer will see if the author chose to ignore analysis results. We still enforce a very low effective false positive rate here (< 10%). Additionally, we only display results for most analyses on *changed lines* by default; this keeps analysis results relevant to the code review at hand.[2] Analyses done at code review time can take longer than analyses that break the build, but the results must be available before the review is over. The mean time for a review of more than one line is greater than 1 hour; we typically expect analyses to complete in less than $5 - 10$ minutes (ideally much less), as developers may be waiting for results.

There are other potential integration points for program analysis. Many IDEs include a variety of static analyses. However, most Google developers do not use IDEs, or do not use IDEs for all tasks [38] – making IDE-only integration untenable. Still, IDE-integration is not precluded as an IDE can issue RPCs to the TRICORDER service. We also leverage testing to run dynamic analysis tools such as ThreadSanitizer [39], [40] or AddressSanitizer [1]. These tools typically have no false positives and < 10x slowdowns. TRICORDER also shows nightly results from some analyses in Google's code search tool as an optional layer. While most developers do not use this feature, it is effective for analyses that have higher false positive rates and have a dedicated cleanup team to sift through the results.

E. Project customization, not user customization

Past experiences at Google showed that allowing user-specific customization caused discrepancies within and across teams, and resulted in declining usage of tools. We observed teams where a developer abandons a tool they were initially using after discovering teammates were committing new instances of code containing warnings flagged by the tool. We

[2] We do show some analyses on all lines by default; as an example, warnings about unused variables can occur on an unchanged line when the block of code using the variable is deleted by the changelist. Developers have the option of also viewing results for unchanged lines during a review if they want to.

have worked to eliminate per-user customization of analysis results.

To achieve this, we got rid of all priority or severity ratings for analysis results. Instead, we try to only show high priority/severity results, and we improve our analysis tools when results are flagged as not important. In cases where there was some debate as to whether an analysis had useful results, we made the analysis optional. Developers still have the ability to explicitly trigger optional analyzers, but they will not run by default. Getting rid of priority ratings had several benefits:

- We were able to remove or improve low-priority checks that had little benefit.
- Instead of having developers filter out analyzer results, we started getting bug reports about broken analyzers. For example, we discovered that the C++ linter was also linting Objective-C files and fixed this issue; previously Objective-C developers had just hidden all linter results.
- We dramatically reduced complaints about why certain results appeared, or why different views were inconsistent.

We do allow limited customization, but the customization is project-based rather than user based. For example, a team can choose to run optional analyses by default on all of their code. We also disable analyzers where they do not apply; for example, we don't run code style checkers on third-party open source code with different code conventions.

IV. Implementation

A. Architecture

To efficiently serve analysis results on changelist creation and editing, TRICORDER leverages a microservices architecture [29]. Thinking in terms of services creates a mindset that encourages scalability and modularity. In addition, TRICORDER is designed with the assumption that parts of the system will go down, which means analysis workers are designed to be replicated and stateless in order to make the system robust and scalable. Analysis results appear in code review as *robot comments* (*robocomments* for short), using the code review tool's commenting system.

Analysis services implement the same API (Section IV-B). This API is defined with protocol buffers [31] as a multi-language serialization protocol, and uses a Google-specific RPC library for communication. TRICORDER includes a series of analyzer worker services written in different languages (Java, C++, Python and Go) implementing this common language-agnostic protocol buffer API. These services provide a language-specific interface for analyzers to implement that abstract away details of handling RPCs; analysis writers can implement analyzers in the language that makes the most sense. TRICORDER also includes compiler-specific analyzer services providing a way to plug into jscompiler, javac, and Clang. We additionally have a binary multiplexor *Linter Worker* service supporting linters written in arbitrary languages.

TRICORDER has three *stages* at which it calls out to analysis services; each stage has successively more information.

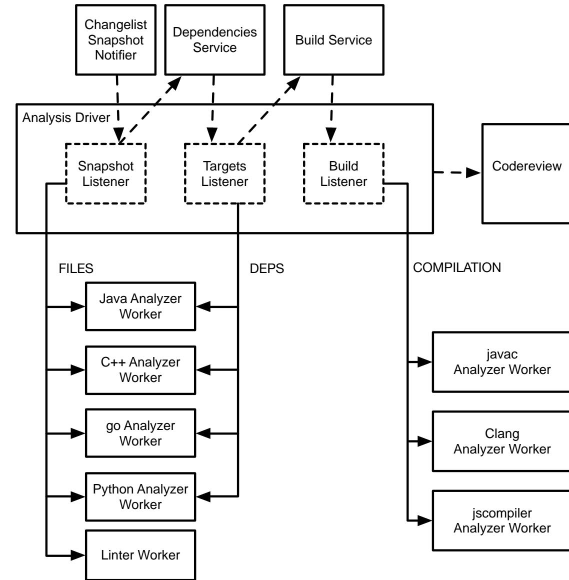

Fig. 1: **Overview of the TRICORDER architecture**. Solid boxes correspond to microservices running as jobs and dashed boxes distinguish separate parts of a job. Solid arrows correspond to RPCs being sent within TRICORDER, while dashed edges refer to RPC to external systems and services.

At the FILES stage, analyzers can access the contents of files that make up the change. For example, linters that check properties like "line is over 80 characters" can run at this stage. At the DEPS (short for build dependencies) stage, analyzers additionally know a list of all the build targets that are affected by the change. For example, an analyzer that reports when a large number of targets are affected can be run at this stage. Finally, at the COMPILATION stage, analyzers have access to the complete AST for the entire program with fully resolved types and implicit expressions. Dividing the analyses into multiple stages has several practical benefits:

- We can provide faster results for analyses at earlier stages, since they do not need to wait for a build.
- We can decrease resource usage by rate limiting analyses at costlier stages.
- Problems with infrastructure we depend on (such as the build service) will not affect analyses from earlier stages.

The overall architecture of TRICORDER is depicted in Figure 1. The main loop of TRICORDER happens in the *Analysis Driver*. The driver calls out to language- or compiler-specific *Analyzer Workers* to run the analyses, then sends results as comments to the code review system. In each worker, incoming analysis requests are dispatched to a set of analyzers. In addition, analysis writers can choose to implement their own standalone service with the analyzer worker API. The analysis driver is divided into separate sections for the different stages of the pipeline: the Snapshot Listener sends requests to FILES analyzers, the Targets Listener sends requests to DEPS analyzers, and the Build Listener sends requests to COMPILATION analyzers. The process is as follows:

1) When a new changelist snapshot is generated, the

changelist snapshot notifier signals to TRICORDER (via a publisher/subscriber model) that there is a new snapshot. This message contains metadata about the snapshot (such as the author, changelist description, and list of files), and a source context (repository name, revision, etc) that can be used to both read the edited files inside the change and post robot comments about the change later. When TRICORDER receives a snapshot notification, it fires off several asynchronous calls:

- TRICORDER sends analyze requests to the FILES analyzers. When it receives results, it forwards the results to the code review system.
- TRICORDER makes a request to the dependencies service to calculate which build targets are affected by the change.[3]

2) The Dependencies Service notifies TRICORDER when it has finished calculating the dependencies, and TRICORDER makes the following asynchronous calls:
   - TRICORDER sends analyze requests to the DEPS analyzers, complete with the list of dependencies transitively affected by the change. When TRICORDER receives analysis results, it forwards them to the code review service.
   - TRICORDER requests that the Build Service starts a build of all targets directly affected by the change.
3) The build service notifies TRICORDER as each independent *compilation unit* is built. The builds are instrumented to capture all inputs needed for each compiler invocation in all supported languages. This set of inputs (e.g. jar files, compiler arguments, headers, etc) is preserved for later use by analyzers.
   When a message arrives that signals a finished compilation unit, TRICORDER sends RPCs to the COMPILATION analyzers. The compiler-specific workers replay the compilation (using the inputs generated during the build) with analysis passes added. COMPILATION analyzers have access to the AST and all other information provided by the compiler. When TRICORDER receives results from the analyzers, they are forwarded to the code review service.

The use of asynchronous communication allows TRICORDER to make more efficient use of its machine resources. Earlier analyses can run in parallel to running a build, and the compilation units are all analyzed in parallel as well. Even the slowest analyses provide results within a few minutes.

B. Plug-in model

TRICORDER supports a plug-in model across languages. Analyzers may be written in any language; currently C++, Java, Python, and Go have the best support. Analyzers may analyze any language, and there are even a variety of analyzers focused on the development process and not a programming language (Section IV-C).

All analyzer services implement the Analyzer RPC API. Most analyzers are running as part of one of the Analyzer Workers and implement a language-specific interface. Each analyzer must support the following operations:

1) `GetCategory` returns the set of categories produced by that analyzer. The category of an analyzer is a unique human-readable name displayed as part of the robocomments it produces.
2) `GetStage` returns the stage in which this analyzer should run.
3) `Analyze` takes in information about the change and returns a list of Notes.

Notes contain key information about the analysis result including:

- The category (and optionally subcategory) of the analysis result.
- The location of the analysis result in the code, e.g. file and range (line/column) within that file.
- The error message.
- A URL with more detailed information about the analyzer and/or the message.
- An ordered list of fixes.

The Notes produced are then posted to Google's internal code review tool as robocomments. Since the structured output is flexible, they may also be used in additional contexts (such as when browsing source code). A more detailed look at this API is available in the open-source version of Tricorder, Shipshape [35]. Shipshape has a different architecture to support the needs of open-source projects, but the API and design was heavily influenced by TRICORDER.

C. Analyzers

Figure 2 shows a selection of 16 analyzers currently running in TRICORDER (there are currently about 30, with more coming online every month).

Six of the analyzers in this table are themselves frameworks. For example, ErrorProne [17] and ClangTidy [11] both find bug patterns based on AST matching for Java and C++ programs, respectively. They each have a variety of individual checks enabled, each implemented as a plugin. Another example is the Linter analyzer. This analyzer is comprised of more than 35 individual linters, all called via a linter binary multiplexor. Linters can be implemented in any language. The multiplexor uses a configuration file to determine which linter to send a particular file to (based on the file extension and path) and how to parse linter output (via a regex). The linter analyzer includes Google-configured versions of popular external linters such as the Java Checkstyle linter [9] and the Pylint Python linter [32], as well as many custom internal linters.

Several TRICORDER analyzers (7 currently) are domain-specific; they are targeted at only a portion of the code

[3] Since the Google codebase is quite large, and projects may have far reaching dependencies, this is important to calculate. For smaller codebases, this stage can likely be skipped in favor of building the entire project.

[4] The AffectedTargets and Builder analyzers are informational, and so do not have a "Please fix" option.

Analyzer	Description	Stage	Impl. Lang.	Analyzed Lang.	# of Plugins	Avg. Results / day	PLEASE FIX Users	NOT USEFUL Users
AffectedTargets	How many targets are affected	DEPS	Java	All		440	-[4]	13
AndoidLint	Scans android projects for likely bugs	COMP.	Java	Android		57	109	71
AutoRefaster	Implementation of Refaster [42]	COMP.	Java	Java	>60	122	3630	670
BuildDeprecation	Identify deprecated build targets	DEPS	Java	Build files		1689	1890	381
Builder	Checks if a changelist builds	COMP.	Java	All		4927	-[4]	491
ClangTidy	Bug patterns based on AST matching	COMP.	C++	C++	>30	4908	5823	1823
DocComments	Errors in javadoc	COMP.	Java	Java		1121	3694	954
ErrorProne	Bug patterns based on AST matching	COMP.	Java	Java	>80	80	2638	206
Formatter	Errors in Java format strings	COMP.	Java	Java		6	561	35
Golint	Style checks for go programs	FILES	Go	Go		1711	1528	486
Govet	Suspicious constructs in go programs	FILES	Go	Go		101	754	156
JavacWarnings	Curated set of warnings from javac	COMP.	Java	Java		288	1631	152
JscompilerWarnings	Warnings produced by jscompiler	COMP.	Java	Javascript		876	733	338
Linter	Style issues in code	FILES	All	All	>35	79079	18675	8316
Unused	Unused variable detection	COMP.	Java	Java		606	5904	833
UnusedDeps	Flag unused dependencies	COMP.	Java	Build files		1419	5018	986

Fig. 2: 16 of 30 analyzers run in TRICORDER. The fourth and fifth columns report the implmenentation language and the target language, respectively. The sixth column reports the number of plugins for analyzers providing an internal plugin mechanism. The seventh column shows the average number of results per day. The final two columns report the number of unique users who clicked on either PLEASE FIX or NOT USEFUL (see Section IV-E) in the year 2014.

base. This includes AndroidLint, which finds both bugs and style violations specifically in Android projects, as well as validators for several project-specific configuration schemas. Several other analyzers (another 7) are about metadata relevant to the changelist. For example, one analyzer warns if a changelist needs to be merged with head, while another warns if a changelist will transitively affect a large percentage of Google's code.

To decide whether an analyzer makes sense to include in TRICORDER, we have criteria for new analyzers, drawn from our experience and philosophy on static analysis (Section III):

1) **The warning should be easy to understand and the fix should be clear.** The problem should be obvious and actionable when pointed out. For example, cyclomatic complexity or location-based fault prediction does not meet this bar.
2) **The warning should have very few false positives.** Developers should feel that we are pointing out an actual issue at least 90% of the time.[5] To measure this, we run analyzers on existing code and manually check a statistically sound sample size of the results.
3) **The warning should be for something that has the potential for significant impact.** We want the warnings to be important enough so that when developers see them they take them seriously and often choose to fix them. To determine this, language-focused analyzers are vetted by language experts.

[5]This 10% false positive threshold matches that used by other analysis platforms such as Coverity [13].

Fig. 3: **Screenshot of analysis results; changelist reviewer view**. In this case there are two results: one from the Java Lint tool (configured version of checkstyle [9]), and one from ErrorProne [17]. Reviewers can click on the NOT USEFUL link if they have a problem with the analysis results. They can also click on PLEASE FIX to indicate that the author should fix the result. They can also view the attached fix (Figure 4).

4) **The warning should occur with a small but noticeable frequency.** There is no point in detecting warnings that never actually occur, but if a warning occurs too frequently, it's likely that it's not causing any real problems. We don't want to overwhelm people with too many warnings.

Analysis developers can try out new analyses on a small set of whitelisted users first who have volunteered to view experimental results. Some analyses can also be run in a MapReduce over all existing code to check false positive rates before being deployed.

//depot/google3/java/com/google/devtools/staticanalysis/Test.java

```
package com.google.devtools.staticanalysis;

public class Test {
  public boolean foo() {
    return getString() == "foo".toString();
  }

  public String getString() {
    return new String("foo");
  }
}
```

```
package com.google.devtools.staticanalysis;

import java.util.Objects;

public class Test {
  public boolean foo() {
    return Objects.equals(getString(), "foo".toString());
  }

  public String getString() {
    return new String("foo");
  }
}
```

Apply Cancel

Fig. 4: Screenshot of the preview fix view for the ErrorProne warning from Figure 3

D. *Fixes*

Two common issues with analysis tools are that they may not produce actionable results, and that it takes effort for busy developers to fix the issues highlighted. To address these problems, we encourage analysis writers to provide *fixes* with their analysis results. These fixes can be both viewed and applied from within the code review tool. Figure 3 shows an example comment produced by TRICORDER. The fix is visible after clicking the "show" link (Figure 4). Note that the fixes here are not tied to a particular IDE, they are part of each analysis and are language-agnostic. Having analyzers supply fixes has several benefits: fixes can provide additional clarification for analysis results, being able to apply the fix directly means that dealing with analysis results does not entail changing context, and tool-provided fixes lower the bar to fixing issues in code.

In order to apply fixes, we leverage the availability of a system that makes the content of a changelist available for edits. That is, edits, like applying fixes, can be made directly to the code in the changelist and the changed code will appear in the workspace of the changelist owner. To implement something similar with Gerrit, one could leverage existing APIs to apply fixes as a patch to the code under review.

E. *Feedback*

In order to respond quickly to issues with analyzers, we have built in a feedback mechanism that tracks how developers interact with the robocomments TRICORDER generates in code reviews. As seen in Figure 3 and Figure 4, there are four links that developers can click:

- NOT USEFUL gives the developers the opportunity to file a bug about the robocomment.
- PLEASE FIX creates a review comment asking the author to fix the robocomment and is only available to reviewers.
- PREVIEW FIX ("show" in Figure 3) shows a diff view of the suggested fix and is available when a robocomment comes with a fix.
- APPLY FIX ("Apply" in Figure 4) applies the suggested fix to the code and is available only to authors when a robocomment comes with a fix. This option can only be seen after using PREVIEW FIX.

We define the "not-useful rate" of an analyzer as:
NOT USEFUL/(NOT USEFUL + PLEASE FIX + APPLY FIX)

Analysis writers are expected to check these numbers through a dashboard we provide. A rate $\geq$ 10% puts the analyzer on probation, and the analysis writer must show progress toward addressing the issue. If the rate goes above 25%, we may decide to turn the analyzer off immediately In practice, we typically work with the analyzer writers to fix the problem instead of immediately disabling an analyzer. Some analyzers only affect a small percentage of developers and so we are not as concerned about a temporary increase in false positives. Nonetheless, having a policy in place has proven invaluable in making expectations clear with analyzer writers.

V. RESULTS

a) Usability: We measure the usability of TRICORDER through the not-useful click rates and numbers of clicks of each type, both for TRICORDER as a whole and for specific analyzers.

Figure 2 shows that developers actively engage with the analyses through clicks. Figure 2 lists the number of unique users in 2014 who clicked on either PLEASE FIX or NOT USEFUL at least once for each analyzer. As can be seen, the Linter has received PLEASE FIX clicks from over 18K users in 2014. The number of people who have every clicked NOT USEFUL is substantially lower across all categories.

Our click rates show that developers are generally happy with the results from the static analysis tools. Figure 5 shows the week-over-week across all of the TRICORDER analyzers; in recent months it is typically at around 5%. When we remove analyzers that are on probation, the number goes down to under 4%.

Figure 6 shows a comparison of the not-useful rates for several Java analyzers. The Checkstyle, ErrorProne, and UnusedDeps analyses are all fairly stable and have a low not-useful rate; the data for ErrorProne has more variance due to the fewer number of findings that it produces. DocComments is more interesting; this analyzer was initially on probation and the developer worked to get the not-useful rate under 10%. However, there was a bug introduced in week 24, which resulted in a sharp increase in NOT USEFUL clicks. The developer was able to use our provided bug reports and click logs to identify the source of the problem, the fix was finally released in week 33.

Analysis writers can also investigate the raw numbers of clicks each week. Figure 7 shows the breakdown for the type of clicks for the ErrorProne analysis. It is interesting to note how correlated PLEASE FIX is with PREVIEW FIX; we hypothesize that many times, a reviewer clicks PLEASE FIX, and

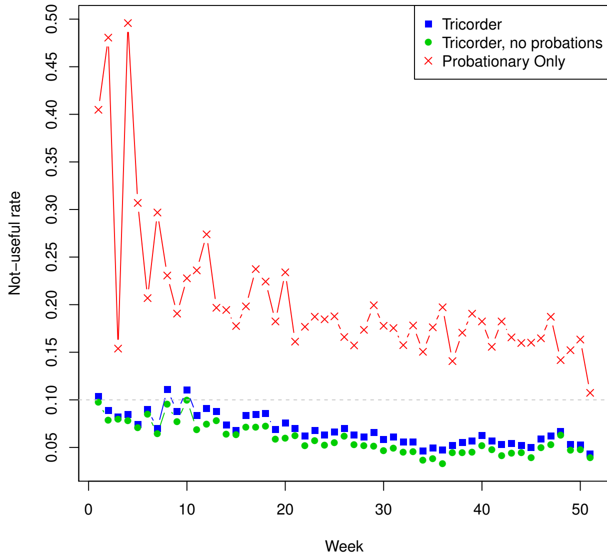

Fig. 5: Not useful click rate for all of TRICORDER, by week for 2014. The probationary analyzers are have either been turned off for having a high not useful rate, or are being actively improved upon. While the have a high rate, there are few enough of them relative to the rest of the analyzers that they have only minor effect on the total rate.

then the author clicks PREVIEW FIX to see what the problem was. The APPLY FIX clicks are much lower. Based on developer observations, we hypothesize that many authors choose to fix the code in their own editor rather than use the code review tool for this purpose, especially if they are already addressing other reviewer comments in their editor. Notice that providing a fix has two purposes; one is to make it easy for developers to apply the fix, but the other is as a further explanation of an analysis result.

Clicks are an imperfect measure of analyzer quality:

- Many developers report fixing issues before their reviewer sees them, so PLEASE FIX counts are known to be low.
- Many developers report fixing issues in their own editor, rather than via APPLY FIX, so those counts are also low.
- Developers may ignore findings they do not plan to fix, rather than clicking NOT USEFUL. This may be a signal only of how strongly the developer feels about the issue.
- Developers may click on NOT USEFUL by accident.

Despite these drawbacks, clicks have been a good signal for "developer annoyance". Our most successful analyzers have not-useful rate between 0-3%.

When a developer clicks on NOT USEFUL, a link appears to file a bug in our issue tracking system against the project responsible for that analyzer; the bug is pre-populated with all the necessary information about the robocomment and gives the developer an opportunity to comment on why they clicked NOT USEFUL. The rate of filing bugs per NOT USEFUL clicks is between 10-60%, depending on the analyzer.

b) Codebase Impact: TRICORDER reduces the number of instances of violations in the codebase over time. For the ClangTidy analyzer, we are able to see that showing results in code review not only prevents new problems from entering the codebase, but also decreases the total number of problems as people learn about the new check. Figure 8 shows the number of occurrences of the ClangTidy `misc-redundant-smartptr-get` check per week; this check identifies cases where there is an unneeded ".get()" call on a smart pointer. The vertical line is when this check began being shown in TRICORDER. After adding the check to TRICORDER the number of instances in the codebase decreased dramatically as developers recognized the inappropriate coding pattern, stopped making such mistakes in new code, and fixed existing occurrences elsewhere in the code. Figure 8 also shows a sampling of trend lines for other ClangTidy fixes; they all show a similar pattern of drop-off or level-off after the check is enabled in TRICORDER. The checks which only level-off are typically checks with more complicated, non-local fixes.

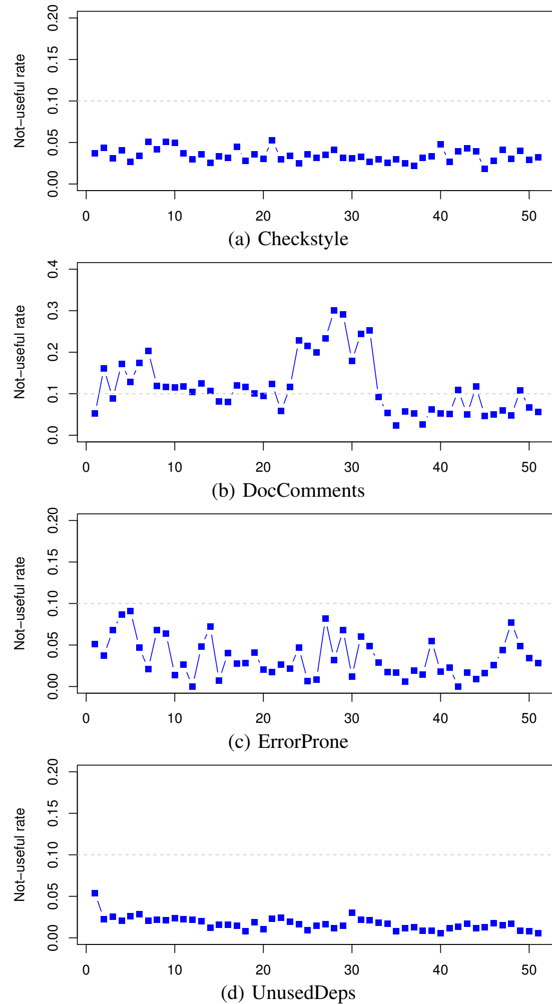

Fig. 6: Weekly not-useful rate for a selection of Java-based analyzers in 2014.

c) Pluggability: TRICORDER is easy to plug into. We evaluate this by demonstrating that a variety of analyses have successfully plugged in. As discussed in Section IV-C, more

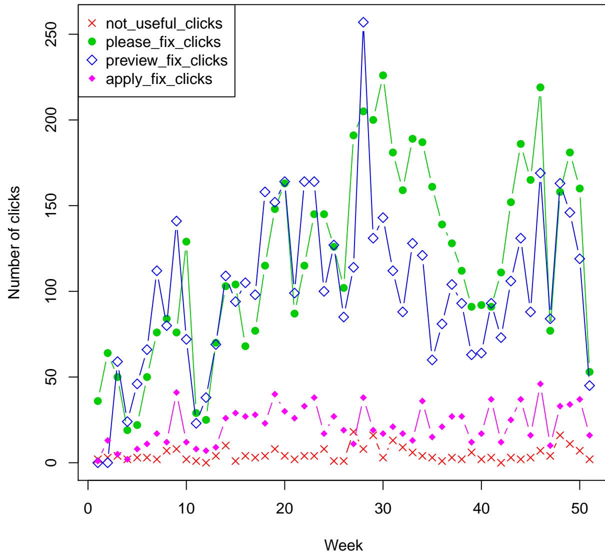

Fig. 7: Weekly clicks, by type, for the ErrorProne analyzer.

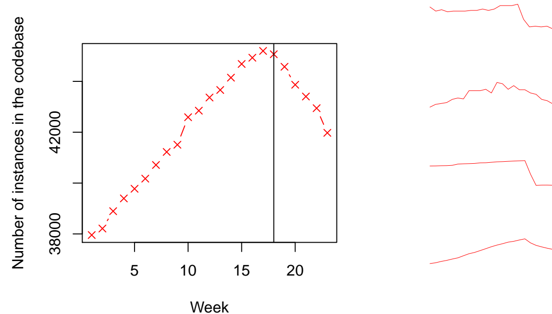

Fig. 8: Instances in the codebase of violations of several ClangTidy checks. The large graph shows the number of violations in the codebase of the check misc-redundant-smartptr-get, the smaller graphs show similar trends for other ClangTidy Checks.

than 30 different analyses are running in TRICORDER. Figure 2 shows a selection of 16 TRICORDER analyzers; these analyzers span a breadth of languages and problems highlighted. 13 of the 16 analyzers in this table (all except Formatter, Build-Deprecation, Builder) were contributed by members of more than 10 other teams. Many additional developers contributed plug-ins to analyzers such as ErrorProne, ClangTidy, or the Linter.

d) Scalability: TRICORDER scales to a very large codebase. To evaluate this, we are running TRICORDER against all changelist snapshots produced each day at Google. Figure 9 shows the average, median, and max counts of several scaling metrics for TRICORDER.

On an average day we run TRICORDER on 31K snapshots, each with an average size of 12 files, and we report findings for several categories and languages. In contrast, ReviewBot [7] was evaluated on 34 review requests (corresponding to change lists), compared to the millions we have used to evaluate TRICORDER. On average, each snapshot contains files from one language, with a max of 22 languages. In total for one day, we

	Average	Median	Max
TRICORDER runs/day	31K	38K	66K
Findings/day	93K	127K	183K
Builds/day	4K	6K	9K
Analyzer runs/day	81K	93K	208K
Files/CL	12	1	333K
Languages/CL	1	1	22
Findings/CL	14	1	5K
"Please fix"/CL	2	1	81
"Not useful"/CL	0.14	0	20
"Please fix"/day	716	907	1786
"Not useful"/day	48	52	123

Fig. 9: Scalability numbers for TRICORDER, in terms of per day or per changelist (CL), collected over 90 days.

report an average of 93K findings for around 30 categories. TRICORDER also runs close to 5K builds per day. Each day, reviewers click PLEASE FIX on an average of 716 findings (416 from the Linters), but only 48 findings get a not-useful click. An average CL has two please-fix clicks and no not-useful clicks. Even though we are producing considerably more findings than are clicked on, most are not actually shown in the review as they appear on unchanged lines.[6]

VI. RELATED WORK

A previous study investigated why developers do not use static analysis tools to find bugs, collecting results from interviews with 20 developers [23]. Many of the conclusions of this study match our experiences with experimenting with various program analysis tools. TRICORDER addresses the main pain points identified by this study. For example, TRICORDER continuously ensures that tool output improves and results are understandable by maintaining a tight feedback loop. TRICORDER also enables collaboration around the tool results by code review integration; reviewers can suggest or comment on static analysis results. This study also highlighted the importance of workflow integration, a main design point of TRICORDER.

There is a wide breadth of research on static analysis tools; we can only describe a portion of the existing tools here. FindBugs [18] is a heuristic-based bug finding for tool that runs on Java bytecode. Coverity [12], Klocwork [24], and Semmle [37], [14] are commercial tools focused on static analysis. Linting tools such as Checkstyle (java) and Pylint (Python) are primarily focused on style issues (spacing, where to break lines, etc) [9], [32].

Analysis results, along with quick-fixes, are also often surfaced in IDEs such as Eclipse [41] and Intellij [22]. We could surface TRICORDER results here too by calling out to our service from within the IDE. Many commercial tools also show results in a dashboard format [12], [37], [24]; while we do have dashboards, we found that this was only useful for analysis writers to help them improve the quality of the analyzers.

[6]Unfortunately, we do not have the ability to measure how many results are viewed by developers.

Other large companies, such as Microsoft are actively experimenting with ways to integrate static analysis into developer workflow [25]. Ebay has developed methods for evaluating and comparing the value of different static analysis tools under experimentation [21]. IBM has also experimented with a service-based approach to static analysis [30]. This work was evaluated through a pilot and surveys with several teams at IBM, and they identified several areas for improvement. TRICORDER addresses each of these improvements: it is integrated into the developer workflow (instead of running in a batch mode), it is pluggable and supports analyses written by other teams, and it includes a feedback loop to analysis writers to improve the platform and analyzers.

One previously published system, called ReviewBot, showed static analysis results in code review [6], [7]. ReviewBot differs from TRICORDER in that reviewers have to explicitly call ReviewBot, analysis results are not shown to reviewers, only three (Java) static analysis tools are currently supported. ReviewBot also added the ability to provide fixes; this fix system is completely decoupled from the analysis results themselves. In contrast, TRICORDER fixes are produced by analyzers, and our mechanism for applying fixes (from structured analysis results) is language-agnostic.

Unlike prior work, we evaluated TRICORDER during the developer workflow, not as a survey of results generated and seen by developers outside of their work environment. This evaluation style allows us to see how developers react in practice, rather than in a lab setting.

VII. DISCUSSION

TRICORDER has been deployed in production since July 2013. Through implementing, launching and monitoring TRICORDER, we have learned several interesting things about how to make program analysis work. In Section III, we outlined our philosophy both in terms of goals for our system and lessons learned from past experiences. We now revisit our goals and the presented evaluation.

Make data-driven usability improvements. In the end, developers will decide whether an analysis tool has high impact and what they consider a false positive to be. **Developers do not like false positives.** This is why it is important to listen to feedback from developers and to act on it. An average of 93K findings per day receive an average of 716 PLEASE FIX and 48 NOT USEFUL clicks. We pay close attention to these clicks, and if needed we put analyzers on probation. We discuss improvements with analyzer writers and encourage them to improve their analyzer – an improvement that may be as simple as updating the wording of results.

Empower users to contribute. TRICORDER does this by providing a pluggable framework which enables developers to easily contribute analyses within their area of expertise. In fact, the majority of all analyses running in TRICORDER are analyses contributed by developers outside the team managing TRICORDER itself.

Workflow integration is key. As with a previously presented tool from VMWare [7], we have found that **code review is an excellent time to show analysis results**. Developers receive feedback before changes are checked in, and the mechanism for displaying analysis results is uniform – no matter which IDE or development environment was previously used. There is also peer accountability as reviewers can see and respond to analysis results; reviewers click PLEASE FIX an average of 5117 times each week.

Project customization, not user customization. Based on experience, we have found that customization at project level is most successful, compared to customization down to user level. This provides flexibility so that teams of developers can have a joint approach to how the code for a project should be developed, and avoids disagreements about analysis results between developers which may lead to results being ignored.

In addition to the above, some things should be mentioned about sophistication, scalability and fixes. All of the checks described in Figure 2 are relatively simple. We are not using any control or data-flow information, pointer analysis, whole-program analysis, abstract interpretation, or other similar techniques. Nonetheless, they find real problems for developers and provide a good payoff. That is, there is **big impact for relatively simple checks**.[7] In general, we have been more successful with analyses that provide a suggested fix. Analysis tools should **fix bugs, not just find them**. There is less confusion about how to address the problem, and the ability to automatically apply a fix provided by the tool reduces the need for context switches. Finally, to ensure analyses can run even at Google's scale, **program analysis tools should be shardable**. Think about an analysis tool as something to map-reduce across large sets of programs.

Final thoughts. In this paper, we presented a static analysis platform, but also our philosophy on how to create such a platform. It is our goal to encourage analysis writers to consider this philosophy when creating new tools and all the tradeoffs of their tool, not just the technically-defined false positive rate or the speed of the analysis. We also encourage other companies, even if they have tried program analysis tools before, to try again with this philosophy in mind. While we also failed to use static analysis tools widely for many years, there is a large payoff for finally getting it right.

ACKNOWLEDGEMENTS

Special thanks to all the Google developers that contributed to make Tricorder a reality, including Alex Eagle, Eddie Aftandilian, Ambrose Feinstein, Alexander Kornienko, and Mark Knichel, and to all the analyzer writers that continue to ensure the platform is successful.

[7]To be clear, we believe more sophisticated static analysis results are also worthwhile (these are typically over-represented in the literature on program analysis).

References

[1] AddressSanitizer Team. AddressSanitizer. http://clang.llvm.org/docs/AddressSanitizer.html, 2012.

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible compiler. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, 2012.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[4] N. Ayewah and W. Pugh. The Google FindBugs fixit. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 241–252, 2010.

[5] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2007.

[6] V. Balachandran. Fix-it: An extensible code auto-fix component in review bot. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 167–172, Sept 2013.

[7] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 931–940, Piscataway, NJ, USA, 2013. IEEE Press.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, , and D. Engler. A few billion lines of code later. *Communications of the ACM*, 53(2):66–75, 2010.

[9] Checkstyle Java Linter. http://checkstyle.sourceforge.net/, August 2014.

[10] Clang compiler. http://clang.llvm.org, August 2014.

[11] ClangTidy. http://clang.llvm.org/extra/clang-tidy.html, August 2014.

[12] Coverity. http://www.coverity.com/.

[13] Coverity 2012 scan report. http://www.coverity.com/press-releases/annual-coverity-scan-report-finds-open-source-and-proprietary-software-quality-better-than-industry-average-for-second-consecutive-year/. Accessed: 2015-02-11.

[14] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-Oriented Queries Made Easy. In *GTTSE*, pages 78–133, 2007.

[15] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[16] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217(0):5 – 21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008).

[17] error-prone: Catch common Java mistakes as compile-time errors. http://code.google.com/p/error-prone/, August 2014.

[18] FindBugs. http://findbugs.sourceforge.net/.

[19] Gerrit code review. http://code.google.com/p/gerrit/, August 2014.

[20] Google. Build in the cloud: How the build system works. Available from http://http://google-engtools.blogspot.com/2011/08, August 2011. Accessed: 2014-11-14.

[21] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, pages 963–970, New York, NY, USA, 2007. ACM.

[22] Jetbrains. IntelliJ IDEA. Available at `http://www.jetbrains.com/idea/`, 2014.

[23] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[24] Klocwork. http://www.klocwork.com/.

[25] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting software. *Software, IEEE*, 21(3):92–100, May 2004.

[26] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *International Conference on Software Engineering (ICSE)*, pages 492–501, New York, NY, USA, 2006. ACM.

[27] L. Layman, L. Williams, and R. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 176–185, Sept 2007.

[28] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. W. Jr. Does bug prediction support human developers? findings from a google case study. In *International Conference on Software Engineering (ICSE)*, 2013.

[29] J. Lewis and M. Fowler. Microservices. http://martinfowler.com/article/microservices.html, 25 March 2014.

[30] M. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *International Conference on Software Engineering (ICSE)*, volume 2, pages 99–108, May 2010.

[31] Protocol Buffers. http://code.google.com/p/protobuf/, August 2014.

[32] Pylint python linter. http://www.pylint.org/, August 2014.

[33] J. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *International Conference on Software Engineering (ICSE)*, pages 341–350, 2008.

[34] C. Sadowski and D. Hutchins. Thread-safety annotation checking. Available from `http://clang.llvm.org/docs/LanguageExtensions.html#threadsafety`, 2011.

[35] C. Sadowski, C. Jaspan, E. Soederberg, C. Winter, and J. van Gogh. Shipshape program analysis platform. https://github.com/google/shipshape. Accessed: 2015-2-11.

[36] C. Sadowski and J. Yi. How developers use data race detection tools. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2014.

[37] Semmle. https://semmle.com/.

[38] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge. Programmers' build errors: a case study (at google). In *International Conference on Software Engineering (ICSE)*, pages 724–734, 2014.

[39] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[40] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler. In *International Workshop on Runtime Verification (RV)*, 2011.

[41] The Eclipse Foundation. The Eclipse Software Development Kit. Available at `http://www.eclipse.org/`, 2009.

[42] L. Wasserman. Scalable, example-based refactorings with refaster. In *Workshop on Refactoring Tools*, 2013.

[43] J. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Always learning. Addison-Wesley, 2012.