Who Broke the Build? Automatically Identifying Changes That Induce Test Failures In Continuous Integration at Google Scale

Celal Ziftci
Google Inc.
ziftci@google.com

Jim Reardon
Google Inc.
jimr@google.com

***Abstract*—Quickly identifying and fixing code changes that introduce regressions is critical to keep the momentum on software development, especially in very large scale software repositories with rapid development cycles, such as at Google. Identifying and fixing such regressions is one of the most expensive, tedious, and time consuming tasks in the software development life-cycle. Therefore, there is a high demand for automated techniques that can help developers identify such changes while minimizing manual human intervention. Various techniques have recently been proposed to identify such code changes. However, these techniques have shortcomings that make them unsuitable for rapid development cycles as at Google. In this paper, we propose a novel algorithm to identify code changes that introduce regressions, and discuss case studies performed at Google on 140 projects. Based on our case studies, our algorithm automatically identifies the change that introduced the regression in the top-5 among thousands of candidates 82% of the time, and provides considerable savings on manual work developers need to perform.**

***Keywords*-Software debugging, Software fault diagnosis, Software testing, Software tools, Software changes, Ranking**

I. Introduction

Google's source code is big, on the order of 2 billion lines of code (LOC) and it evolves rapidly [1], [2]. On average, 40000 code changes are submitted to the Google code repository every day, and 15 million lines of code in 250000 files are modified every week [2].

To make such rapid development and evolution more reliable, Google has adopted Continuous Integration (CI) [3], where each code change triggers code to be compiled and tests to be executed so that regressions can be caught earlier in the development lifecycle [4].

Regressions have many causes, e.g. code does not compile, tests fail, performance of the system drops. This paper specifically addresses regressions where code compiles, but there are test failures.

Figure 1 shows the code repository and development workflow at Google. The Google code repository does not use multiple branches, i.e. there is a single branch for the entire codebase, called `HEAD`, where all developers submit their changes in total order [2]. Developers change the code in the repository by submitting atomic changes to the codebase, called a *changelist* (CL). *System version* refers to the version of `HEAD` as it is after a CL is submitted. So every CL submission results in a new system version.

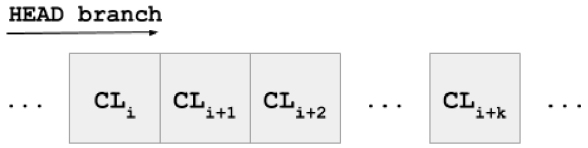

Figure 1: Code repository and development workflow at Google. Changes are submitted to the code repository in atomic units called changelists (CLs). Each CL is submitted to the same code repository branch called `HEAD`.

Each CL is validated by executing tests. Some tests are run before the CL is submitted, while some tests are run after the submission. When a test executed after CL submission fails, the CL is said to have induced a regression. When a CL introduces such a regression into `HEAD` that results in test failures, it is important to identify it quickly, so that the development velocity is not disrupted. As an example, assume a regression is introduced to the codebase by changelist CL_i in Figure 1, and some tests fail at all system versions after that CL. If a developer decides to debug the failing tests to locate their root cause, she can use existing debugging techniques in the literature on any failing version of the system (any version including and after CL_i). However, it is critical to identify that CL_i was the changelist that introduced the regression, because:

- The source code modifications in CL_i might provide clues about the root cause of the regression, as it introduced the regression in the first place.
- Debugging the failing tests at version CL_i is more beneficial than debugging at some later version, say CL_{i+k}, since changes after CL_i increase/change the code to be analyzed and the developer can be misled.
- More faults might have been introduced at later versions of the system, which makes it harder to debug and identify root causes of a regression.
- Evidence suggests that the author of CL_i might locate and fix the fault easier than any other developer [5].
- Reverting the change that introduces the regression, such as CL_i, is a common practice to resolve the

978-1-5386-2717-4/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-SEIP.2017.13

Table I: Classification of tests at Google according to their size

	Runtime Limit	Example Tests
Small	1 min	Unit
Medium	5 mins	Unit / Integration
Large	15 mins	System / Integration / End-to-end
Enormous	No limit	System / Integration / End-to-end

regression quickly [6].

In summary, for code repositories where there is rapid evolution as at Google, finding the system version where a regression was first introduced is the crucial first step in debugging failing tests. In this paper:

- We propose a novel algorithm that can automatically identify changes that introduce regressions into the codebase,
- We evaluate our algorithm on a case study we ran on 140 projects in Google and discuss the results.

II. Continuous Integration at Google

As summarized in Table I, tests are classified into four categories at Google according to their **size**: small, medium, large, enormous [7]. Size refers to the execution time of the test, excluding any overhead of setting up the test, e.g. compilation.

At Google, given the size of each test in Table I, there are two typical development workflows regarding CLs where tests are executed (configured by developers manually, and enforced by infrastructure automatically):

1) **Presubmit tests**: Tests are run before a CL is submitted. Unless all tests pass, the CL is not allowed to be submitted. Typical presubmit tests are small/medium tests (although large tests are also allowed), so that developers are not blocked for a long time to get the results of test execution before submitting a CL [2].
2) **Postsubmit tests**: A CL is submitted to the repository without waiting on the results of test execution. Tests are executed periodically to check if any of the submitted CLs caused a regression, and if so, developers are notified of the regression after the submission. Typical postsubmit tests are large/enormous tests, because such tests would take too long to execute and hinder development velocity if they were to be executed before CL submission.

When a presubmit test fails, the CL is not allowed to be submitted. Therefore, there is no possibility of introducing a regression due to test failures. However, this does not hold for postsubmit tests. Such tests do not block CL submission, and regressions can surface after a certain time period.

Figure 2 shows an overview of how such tests can fail after some CLs are submitted. The CI system executes tests periodically (e.g. every 10 minutes or every N CLs). In this case, it has executed the tests at versions CL_G (green CL) with all tests passing and CL_R (red CL) with some

Figure 2: CL_G denotes the version of HEAD when tests were passing. CL_R denotes the version of $HEAD$ when tests are failing. Tests have only been executed at versions CL_G and CL_R, but not in between. Any CL submitted within $(CL_G, CL_R]$ can be the CL that caused the tests to fail, called the **culprit CL**.

tests failing. Given this, some ***culprit CL*** in the range $(CL_G, CL_R]$ must have caused a regression.

A well-known solution to finding the culprit CL for postsubmit tests in literature is performing a search over the CL range $(CL_G, CL_R]$. An example is binary search as done in "git bisect" [8], where the tests are executed at the CL $\frac{CL_G+CL_R}{2}$. If tests are failing at that version of the system, the search is carried out between $(CL_G, \frac{CL_G+CL_R}{2}]$. Otherwise, it is carried out between $(\frac{CL_G+CL_R}{2}, CL_R]$. This procedure continues recursively, until the culprit CL is identified. To obtain the result faster, an n-ary search, instead of binary, can also be used, but the idea still holds.

Searching as discussed above is a possible solution for small/medium sized tests, as they don't take too long to execute (5 minutes is the upper limit). For instance, if there are 1000 CLs in the search window, the culprit CL can possibly be identified within 10 iterations, hence 50 minutes.

However, for large and enormous tests, this approach quickly becomes undesirable:

- If a test takes 45 minutes to run, it would potentially take 7.5 hours to find the culprit CL in a 1000 CL search window,
- If code development continues in the presence of failing tests (such as at Google), other developers might introduce more regressions along the way before the current regression is resolved,
- Re-running tests will use a large amount of resources that can be used for other purposes.

Therefore, identifying the culprit CL quickly, resolving the regression, and getting back to green is important to keep development momentum. This paper focuses on solving the problem of finding such culprit CLs that cause regressions for large and enormous postsubmit tests. The technique in this paper focuses on solving this problem such that:

- Regressions are automatically detected without any human intervention or input (e.g. creation of a bug report),
- A list of likely culprit CLs is provided to developers rapidly,
- Tests are not re-executed to conserve resources.

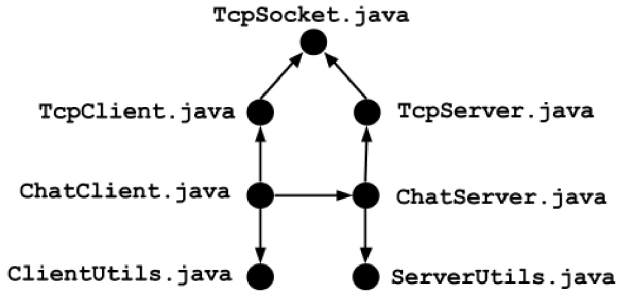

Figure 3: Files and file level dependencies for a simple chat system.

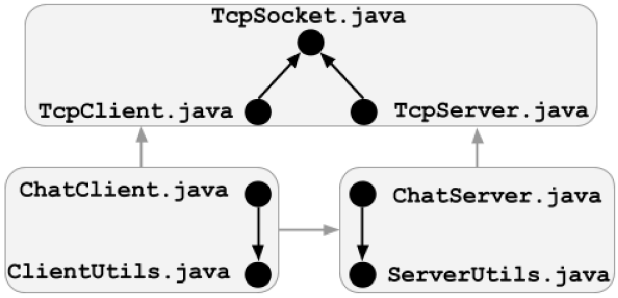

Figure 4: Targets and target level dependencies for the chat system.

III. Build System at Google

In this section, we discuss the build system used at Google, called *blaze* [9] (partially open sourced as *bazel* [10]), and define terms used to describe our algorithm in the next section.

At the lowest level, the build system at Google is composed of files `f` that can have dependencies on each other. As an example, Figure 3 depicts files and their dependencies for a simple chat system in Java.

Files can be grouped into logical units. For the chat system, there are three logical groups: server side code composed of `ChatServer.java` and `ServerUtils.java`, client side code composed of `ChatClient.java` and `ClientUtils.java`, and finally network communication related code in `TcpClient.java`, `TcpServer.java` and `TcpSocket.java`. Such logical groups are called build targets.

Definition 1. *A build target T is a logical grouping of files (e.g. test code, source code, data) in the code repository that explicitly lists the files it encapsulates and its dependencies.*

Figure 4 shows the files and the build targets that encapsulate them for the chat system: network, server and client code. Targets also depend on each other based on the required file level dependencies. As an example, the target for `ChatClient.java` depends on the target for `TcpClient.java`.

```
java_library(
  name = "network_lib",
  srcs = [
    "TcpClient.java",
    "TcpServer.java",
    "TcpSocket.java",
  ],
)

java_library(
  name = "chat_server",
  srcs = [
    "ChatServer.java",
    "ServerUtils.java",
  ],
  deps = [
    ":network_lib",
  ],
)

java_library(
  name = "chat_client",
  srcs = [
    "ChatClient.java",
    "ClientUtils.java",
  ],
  deps = [
    ":network_lib",
    ":chat_server",
  ],
)

java_binary(
  name = "chat_system",
  srcs = [
    ":chat_client",
    ":chat_server",
  ],
)
```

Figure 5: `BUILD` file with target definitions for the chat system.

Build targets are defined in `BUILD` files. Figure 5 shows the definitions of the targets for the chat system: `network_lib`, `chat_client` and `chat_server`; and a target that is an executable binary: `chat_system`. As expected, dependencies are specified for each build target: e.g. `chat_server` depends on `network_lib`.

Definition 2. *Build graph (or build tree) B is a directed acyclic graph (DAG) where nodes T are build targets and there is an edge from T_i to T_j if T_i depends on T_j.*

As shown in Figure 6, targets in the entire code repository comprise the build graph. Below are functions that denote relationships of files and targets.

Definition 3. *S^{FT} denotes the set of targets that a file belongs to in the build tree:*

$$S^{FT}(f_i) = \{T : f_i \text{ is listed as a source file in } T \in B\}$$

Typically, a file is part of a single target. But it is not uncommon to have multiple targets that include the same file, e.g. there might be a target that exposes fewer features than another target, and both may need the same file.

In the build tree, targets have direct and transitive dependencies on each other.

Definition 4. *$D(T_i)$ denotes the direct and transitive dependencies of a build target in the build tree:*

$$D(T_i) = \{T : \text{ there is a path from } T_i \in B \text{ to } T \in B\}$$

Definition 5. *$d^T(T_i, T_j)$ denotes the length of the shortest path between two build targets (0 if there is no path):*

$$d^T(T_i, T_j) : \text{shortest distance from } T_i \in B \text{ to } T_j \in B$$

IV. Developer Workflow at Google

In this section, we discuss the typical development workflow at Google and define some functions that relate the

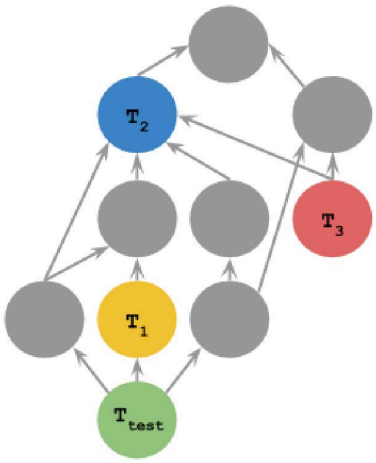

Figure 6: A sample build graph (or build tree) with a test target T_{test} and other non-test targets. $T_1 \in D(T_{test})$, $T_2 \in D(T_{test})$, but $T_3 \notin D(T_{test})$.

workflow to the build system.

Definition 6. *A changelist (CL) is an atomic change to the code repository that can add / update / delete one or more files upon submission.*

A CL can contain any number of files, and upon submission of the CL, the code repository is updated with the files contained in the CL using transactional semantics (i.e. all or nothing).

Definition 7. $S^F(CL_i)$ *denotes the set of files contained (added / updated / deleted) in the changelist* CL_i.

Definition 8. $S^T(CL_i)$ *denotes the set of targets touched by the changelist* CL_i*:*
$S^T(CL_i) = \{T : f \in S^F(CL_i) \wedge T \in S^{FT}(f)\}$

CLs are submitted to a single code repository at Google, called `HEAD` [2], and there is a total order between them.

V. Culprit Finder

Section II discussed the problem of finding the *culprit CL*, the CL that introduced a regression into the code repository. As in Figure 2, if the CI system determines that at system version CL_G all tests were passing, while at system version CL_R some tests are failing, the CLs in the range $(CL_G, CL_R]$ are **suspect CLs**.

In this section, we discuss our algorithm which ranks these CLs according to their **suspiciousness** using a combination of several heuristics, and notifies developers with the list of suspect CLs ordered by suspiciousness to help them identify the culprit CL.

A. Heuristics

1) Changelog: The first heuristic eliminates CLs that cannot possibly be the cause of a test failure. Assume that T_{test} is a build target that contains tests.

Definition 9. $S^L(T_{test}, CL_G, CL_R)$ *denotes the changelog of* T_{test} *between the system versions* CL_G *and* CL_R*, i.e. any CL that touched a target* T_{test} *depends on:*
$S^L(T_{test}, CL_G, CL_R) = \{CL : CL \text{ in } (CL_G, CL_R] \wedge T \in S^T(CL) \wedge T \in D(T_{test})\}$

A CL CL_i can only be a suspect CL if it is in the changelog $S^L(T_{test}, CL_G, CL_R)$, i.e. if and only if it contains at least one file (hence a target) that T_{test} depends on. As an example, assume the build tree looks like the one in Figure 6. A CL that modifies T_1 would be in the changelog of T_{test}, while one that only modifies T_3 would not be.

2) CL Size: The second heuristic uses the size of the CL: given two CLs, the one that contains more files is more suspicious. This heuristic is trivial, since it simply states that a target is more likely to cause a test failure if it touches more code that the failing test target depends on.

3) Target Distance: The final heuristic uses proximity between targets in the build tree. In the build tree in Figure 6, this heuristic proposes that when T_{test} fails, a CL modifying T_1 is more suspicious than one modifying T_2. This is based on two observations in the rapid development cycles at Google: **(i)** A target closer to the root in the build tree, such as T_2, has more dependencies in the entire build tree, hence, if broken, has higher risk of disrupting the development for many developers compared to T_1. Therefore, it is potentially tested better and/or in more depth before the CL is submitted. **(ii)** In the case that a CL submission that changes T_2 does cause a breakage, it is likely to be noticed quicker than a target like T_1. In the sample build tree, T_1 has a single dependency, namely T_{test}. So it depends on T_{test} failing for regressions to be noticed due to any CLs changing T_1. T_2, on the other hand, has many dependencies (direct and transitive). Therefore, if it causes a breakage, any one of those dependencies will likely notice the breakage, and pave the way for a quick fix by notifying the developer that submitted any CLs modifying T_2.

B. Suspiciousness

The heuristics discussed in the previous section are combined into a suspiciousness score for a CL, given a test target T_{test} is failing.

Definition 10. $d^{TW}(T_i, T_j)$ *denotes the weighted distance between two targets* T_i *and* T_j *(0 if there is no path from* T_i *to* T_j*):*

$$d^{TW}(T_i, T_j) = \frac{11}{\sqrt{40 \times \pi} \times exp(\frac{-0.5 \times (d^T(T_i,T_j)-1)^2}{20})}$$

Weighted distance $d^{TW}(T_i, T_j)$ is based on the distance between two targets $d^T(T_i, T_j)$. Keeping T_i constant, the weighted distance between T_i and another target is shown in Figure 7. When $d^T(T_i, T_j) = 1$, the weighted distance $d^{TW}(T_i, T_j) \approx 1$. As the distance between targets increases, the weighted distance rapidly decreases. Using the example

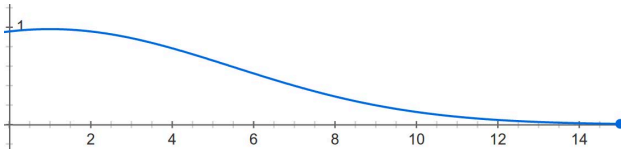

Figure 7: x-axis is distance d^T, y-axis is weighted distance d^{TW}. Value of weighted distance between two targets quickly drops as their distance increases.

in Figure 6:

$$d^T(T_{test}, T_1) = 1 \Leftrightarrow d^{TW}(T_{test}, T_1) \approx 0.98$$
$$d^T(T_{test}, T_2) = 2 \Leftrightarrow d^{TW}(T_{test}, T_2) \approx 0.95$$
$$d^T(T_{test}, T_3) = 0 \Leftrightarrow d^{TW}(T_{test}, T_3) = 0$$

Definition 11. *$r^{CL}(T_{test}, CL_i)$ denotes the suspiciousness score of CL_i given test target T_{test} fails:*
$r^{CL}(T_{test}, CL_i) = \sum_{T_i \in S^T(CL_i)} d^{TW}(T_{test}, T_i)$

Higher suspiciousness scores imply higher likelihood for a CL to have caused the test failure. The suspiciousness function encapsulates the heuristics discussed in the previous section:

- Changelog: In the build tree in Figure 6 $T_3 \notin D(T_{test})$. If a CL only modifies T_3, it will not be in the changelog for T_{test}.
- CL size: A CL that modifies T_1 and T_2 together will have a higher suspiciousness score than a CL that only modifies T_1.
- Distance: A CL that only modifies T_1 will have a higher suspiciousness score than a CL that only modifies T_2.

C. Notification

As in Figure 2, given all tests pass at system version CL_G and some test target T_{test} fails at system version CL_R, culprit finder calculates the suspiciousness score for all CLs in the changelog $S^L(T_{test}, CL_G, CL_R)$ and notifies developers via email with a report that ranks CLs from more suspicious to less suspicious.

A sample report is shown in Figure 8. A developer that receives the email can look at the CLs in the suspect list in the given order to identify the culprit CL (e.g. they typically check if the failing target T_{test} has any relation to the description/files in the CL, or they run the tests at that CL). Once the developer identifies the culprit CL from the suspects list, she can then debug to find the root cause of the regression in the codebase.

VI. Case Studies

In this section, we discuss case studies conducted at Google to investigate:

1) Is Culprit Finder beneficial?
2) Is there any bias in the feedback developers report?

Our case studies are based on the feedback reports submitted by Google developers using the feedback links in the

Your test target $\mathtt{T}_{\mathtt{test}}$ is failing due to a regression introduced between CLs `(1987234, 1992398]`. Here are the suspects:

Rank	CL	Suspiciousness	Feedback
1	1987236	13.456	This CL is the culprit
2	1989147	4.107	This CL is the culprit
3	1991205	0.981	This CL is the culprit

Figure 8: A notification email with suspect CLs sent to developers. Emails contain all CLs in the changelog S^L. After identifying the culprit CL, a developer can click on the respective *'This CL is the culprit'* link to submit feedback.

notification emails shown in Figure 8. We asked developers to investigate the suspect CLs in the list, determine the culprit CL, and click only on the appropriate link labeled *'This CL is the culprit'* next to the culprit CL. Once a developer clicks the link, a feedback report is recorded with information on:

- The test target, T_{test} in Figure 8,
- The CL window, i.e. start and end CLs, $(1987234, 1992398]$ in Figure 8,
- The total number of CLs in the suspect list, i.e. changelog size, 3 in Figure 8,
- The rank of the culprit CL in the suspect list, e.g. 2 for CL 1989147 in Figure 8.

During our case studies: (i) we asked all teams in Google to enable Culprit Finder for large/enormous tests, but it is up to teams to enable it or not; (ii) we generated a feedback report for all test failures for which Culprit Finder is enabled; (iii) the set of CLs in our notification emails is exhaustive, i.e. the culprit CL is always in the changelog; (iv) we requested feedback for all reports, however, this is completely voluntary, and we don't know when developers decide to report feedback; (v) we cannot run test targets automatically to find the actual culprit CL, e.g. using binary search discussed earlier, due to permissions (we may not have permission to run the test due to sensitive data/infrastructure) and resource constraints (tests may be prohibitively large to run repeatedly); (vi) we ran our case study for over a year, the results discussed in this paper are all the reported feedback over the span of the study.

Statistics on the set of feedback reports we received are summarized in Table II. In our experiments, Culprit Finder was enabled for use on 297 unique test targets. We received at least one feedback report for 140 of those targets. In total, we received 377 unique feedback reports (if multiple reports are submitted for the same failure by two developers, we only count the initial one submitted).

Table II: Statistics on the feedback reported by developers

Total unique test targets for which CULPRIT FINDER was enabled	297
Total unique test targets for which any feedback was reported	140
Total unique developers that provided feedback	172
Total feedback reports	377
Min # feedback reports per test target	1
Max # feedback reports per test target	22
Mean # feedback reports per test target	2.69
Median # feedback reports per test target	2.00

Table III: Statistics on the feedback reported by developers separated by control and experiment groups

	Control	Experiment
Total unique test targets for which any feedback was reported	2	138
Total unique developers that provided feedback	2	170
Total feedback reports	34	343
Min # feedback reports per test target	13	1
Max # feedback reports per test target	21	14
Mean # feedback reports per test target	17.00	2.45
Median # feedback reports per test target	17.00	2.00

We received between 1 and 22 reports for each of the test targets. Mean number of feedback reports for each project was 2.69, while the median was 2.

Since the submission of feedback reports was voluntary (e.g. users may report feedback when CULPRIT FINDER is not very successful, or they may report when it ranks their culprit CL high, hence is helpful to a developer), we did another case study by specifically asking certain developers to respond to **all** notification emails for their test targets, without any selection bias. Table III shows a separated view of the data in Table II into two groups: **Control** group consists of developers that respond to all notifications, **experiment** group consists of all other developers, whose behavior we had no control over.

VII. DISCUSSION

We use several metrics to assess the quality of CULPRIT FINDER. Table IV summarizes the mean values on several dimensions collected in the feedback reports.

A. Benefit From Changelog: S^L

In this section, we discuss the benefit obtained by generating changelog S^L upon a test failure.

Definition 12. *$rank_W$ denotes the mean rank of a culprit CL in the CL window between the last green and first red versions:*

$$rank_W(T_{test}, CL_G, CL_R) = \frac{CL_G - CL_R}{2}$$

If a developer were to be given all of the CLs in the CL window $(CL_G, CL_R]$ with no ranking between them, she would need to investigate, on average, half of the CLs in the window until she finds the culprit CL. This is denoted by $rank_W$. For instance, in the example in Figure 8:

Table IV: Statistics on CULPRIT FINDER feedback reports

Mean CL window size	12183.94
Median CL window size	3457.00
Mean changelog size	38.81
Median changelog size	14.00
Mean $rank_W$	6091.97
Mean $rank_L$	19.40
Mean $rank_{CF}$	5.56
Mean b_{S^L}	0.99
Mean $b_{r^{CL}}$	0.36

$rank_W(T_{test}, 1987234, 1992398) = \frac{1992398 - 1987234}{2} = 2582$

Definition 13. *$rank_L$ denotes the mean rank of a culprit CL in the changelog:*

$$rank_L(T_{test}, CL_G, CL_R) = \frac{|S^L(T_{test}, CL_G, CL_R)|}{2}$$

If a developer were to be given all of the CLs in the changelog with no ranking between them, she would need to investigate, on average, half of the CLs in the changelog until she finds the culprit CL. This is denoted by $rank_L$. For instance, in the example in Figure 8:

$$rank_L(T_{test}, 1987234, 1992398) = \frac{3}{2} = 1.5$$

Definition 14. *$b_{S^L} \in [0, 1)$ denotes the benefit provided by the changelog heuristic S^L:*

$$b_{S^L}(T, CL_G, CL_R) = \frac{rank_W(T, CL_G, CL_R) - rank_L(T, CL_G, CL_R)}{rank_W(T, CL_G, CL_R)}$$

The benefit function $b_{S^L} \in [0, 1)$. If it is 0, no benefit was provided. Otherwise, some benefit was provided to the developers. Higher b_{S^L} indicates more benefit.

In our case studies, summarized in Table IV:

- The mean window size is 12183.94,
- The mean changelog size $|S^L|$ is 38.81,
- The mean $rank_W$ is 6091.97,
- The mean $rank_L$ is 19.40.
- The mean b_{S^L} is 0.99.

Therefore, automatically eliminating CLs based on the changelog S^L dramatically narrows down the list of suspect CLs and benefits developers.

B. Benefit From Suspiciousness: r^{CL}

In this section, we discuss the benefit obtained by ranking suspect CLs using the suspiciousness function r^{CL}.

Definition 15. *$rank_{CF}$ denotes the rank of the culprit CL in the changelog reported by* CULPRIT FINDER *in the notification email.*

$rank_{CF}$ is reported by developers as feedback.

Definition 16. *$b_{r^{CL}} \in [-1, 1)$ denotes the benefit provided by the suspiciousness scoring function r^{CL}:*

$$b_{r^{CL}}(T, CL_G, CL_R) = \frac{rank_L(T, CL_G, CL_R) - rank_{CF}(T, CL_G, CL_R)}{rank_L(T, CL_G, CL_R)}$$

The benefit function $b_{r^{CL}} \in [-1, 1)$ is the measure of the amount of manual work a developer is saved due to r^{CL}. If $b_{r^{CL}} = 0$, no benefit was obtained. If $b_{r^{CL}} < 0$, a developer was harmed because she ended up looking through more CLs than she would need to if r^{CL} was not used. If $b_{r^{CL}} > 0$, the

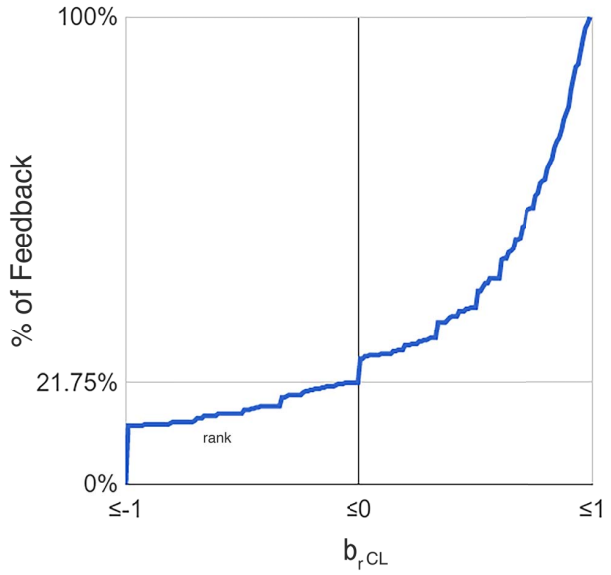

Figure 10: Cumulative benefit values $b_{r^{CL}}$ in reported feedback. 21.75% of the time, suspiciousness heuristic provided a negative benefit, while in 78.25% of the time, it provided zero or positive benefit.

developer benefited from using r^{CL} heuristic. Higher $b_{r^{CL}}$ indicates more benefit.

In our case studies, summarized in Table IV:

- The mean $rank_L$ is 19.40,
- The mean $rank_{CF}$ is 5.56,
- The mean $b_{r^{CL}}$ is 0.36.

As a result, suspiciousness heuristic r^{CL} provides non-negligible benefit on top of changelog S^L generation for developers.

C. Detailed Analysis Of Feedback Reports

In addition to the statistics in Table IV, in this section, we provide more detailed insight into the feedback reports.

Figure 9 shows the percentage of feedback reports with a cumulative distribution of $rank_{CF}$. The chart on the left shows that in 51.99% of the feedback reports, culprit CL was ranked as #1 in the suspect list, and 82.23% of the time, it was in top 5. The chart on the right provides higher resolution data for all $rank_{CF}$ values.

Figure 10 shows the cumulative distribution of benefit $b_{r^{CL}}$ in the feedback reports. In 78.25% of the feedback, r^{CL} provided a positive benefit by decreasing the manual work to be performed by developers.

Finally, Figure 11 displays the benefit for each individual feedback report. In each chart:

- The x-axis is the size of changelog: $S^L = rank_L \times 2$,
- The y-axis is $rank_L - rank_{CF} = b_{r^{CL}} \times rank_L$, a function positively correlated with benefit,

Table V: Statistics on CULPRIT FINDER feedback reports separated by control and experiment groups

	Control	Experiment
Mean CL window size	3267.59	13067.78
Median CL window size	1789.00	3960.00
Mean changelog size	26.97	39.97
Median changelog size	18.50	13.00
Mean $rank_W$	1633.79	6533.89
Mean $rank_L$	13.49	19.99
Mean $rank_{CF}$	3.74	5.74
Mean b_{S^L}	≈ 1.00	0.99
Mean $b_{r^{CL}}$	0.44	0.35

- The continuous line is where: $rank_L = rank_{CF} \Rightarrow b_{r^{CL}} = 0$.

In these charts, if a data point is above the continuous line, that means r^{CL} provided a benefit to the developer. The top-left chart shows the data points for all feedback reports, while the subsequent charts show a zoomed in view of the same data cut at certain thresholds to remove outliers. Overall, in 78.25% of the reports, r^{CL} provides a positive benefit to developers.

D. Comparing Control vs Experiment

In this section, we discuss the results obtained from the two sets of feedback reported: control and experiment. As discussed in the previous section: **control** group consists of developers that respond to all notifications, **experiment** group consists of all other developers, whose behavior we had no control over.

Table V lists the statistics in each set of feedback reports. Based on the numbers:

- Control group had smaller CL window sizes,
- Control group had a smaller mean changelog size,
- Control group had a larger median changelog size,
- Control group had lower ranks,
- Both groups had similar b_{S^L} benefit,
- Control group had higher $b_{r^{CL}}$ benefit.

Since the number of total feedback reports is quite limited in both control and experiment groups, we don't strongly conclude any bias. However, in the experiment group, we observed higher mean $rank_{CF}$ along with lower median changelog size, and lower benefit $b_{r^{CL}}$, which suggests some developer bias where developers tend to report feedback when CULPRIT FINDER is not as successful as they expect and ranks the culprit CL low.

E. CULPRIT FINDER Runtime

In general, there are several factors that affect the runtime of CULPRIT FINDER. CULPRIT FINDER will take longer when: (i) the CL window is large, since it calculates the changelog from the CL window, (ii) the test target is lower in the build tree, since it will potentially have more dependencies, hence more targets to analyze, (iii) the changelog is large, since it calculates the weighted distance for each CL in the changelog, (iv) the CLs in the changelog are large,

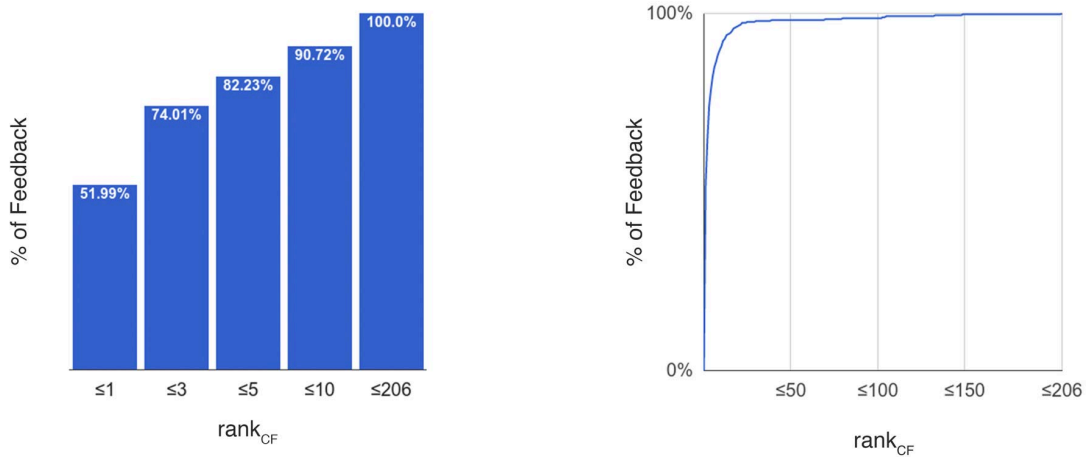

Figure 9: $rank_{CF}$ in feedback reports submitted by developers. The chart on the left shows that in 51.99% of the feedback reports, culprit CL was ranked as #1 in the suspect list, and 82.23% of the time, it was in top-5. The chart on the right provides higher resolution data for all rank values.

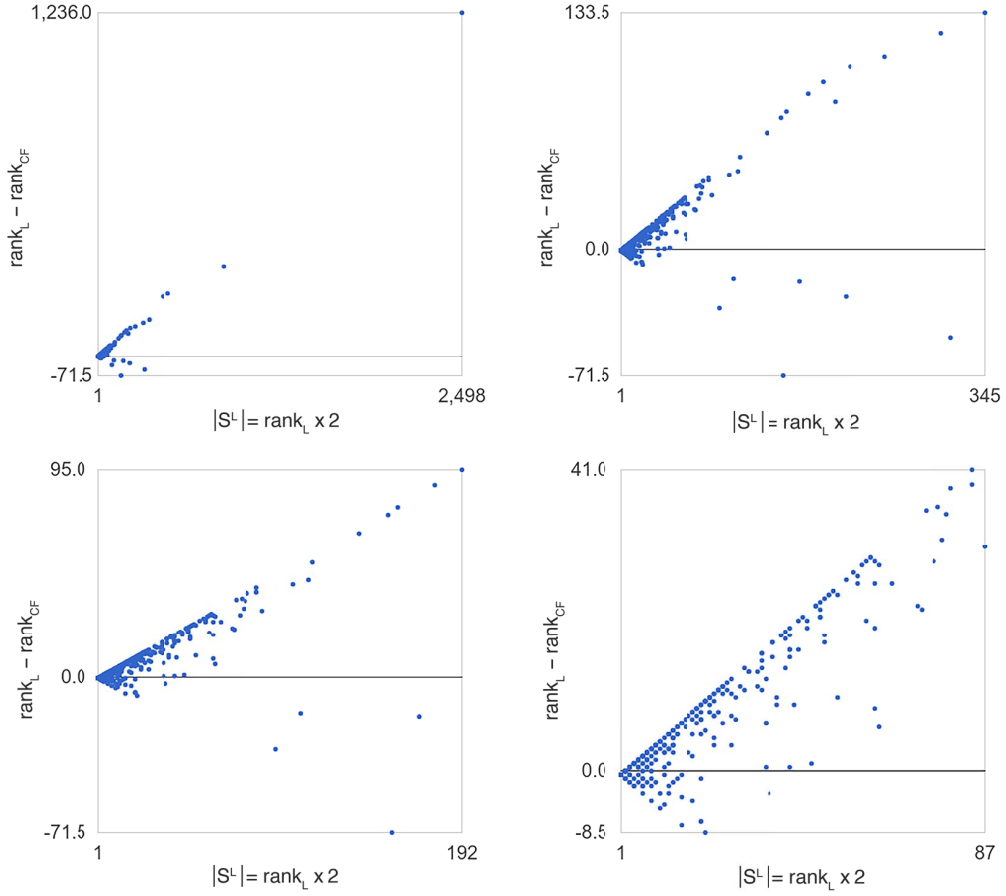

Figure 11: Charts that show $rank_L - rank_{CF}$ against $\left|S^L\right| = rank_L \times 2$. In each chart, the continuous line is 0, i.e. where $rank_{CF} = rank_L$. Data points above the line are where $rank_{CF} < rank_L$, i.e. b_{rCL} resulted in positive benefit to developers. Anything below the line was harmful. Top left chart displays the results for all feedback, while other charts show zoomed in portions of the data for better viewability.

Table VI: Sample runtimes for CULPRIT FINDER

Target	CL Window Size	Changelog Size	Runtime
T_x	839571	2498	17 hours
T_y	182445	857	24 minutes
T_z	13675	60	30 minutes
T_p	4154	26	13 minutes
T_q	651	5	14 minutes

since it calculates weighted distance for each CL and this includes finding the targets for each file in a CL.

Therefore, it is possible to get different CULPRIT FINDER runtimes for different test targets. Table VI lists several sample runtimes of CULPRIT FINDER for different test targets and CL window sizes we have collected manually looking at CULPRIT FINDER execution logs. These times are the elapsed time from the moment a test target is identified to be broken to the time the CULPRIT FINDER notification is received by developers, which includes setup costs of CULPRIT FINDER (downloading the binary, extracting it and starting the process on the Google cloud).

The largest test target failure for which CULPRIT FINDER ran had a CL window size of 839571 and it took 17 hours to run. However, such large CL windows are not typical at Google. The median CL window size in our case study was 3457. Table VI lists a runtime of 13 minutes for a CL window size of 4154, and similar runtimes for sample smaller and larger CL windows.

Based on interviews with developers, these runtimes are within reasonable bounds as they can get a notification email and start investigating the test failure within the same day. Unfortunately, in our case studies, we did not record the runtime information for each CULPRIT FINDER run. Therefore, we don't have any aggregate metrics, and only provide the sample runtimes in Table VI.

F. Threats To Validity

There are several threats to the validity of our results.

Test targets: Over the span of the study, we received feedback for 140 different test targets, although there were 297 for which CULPRIT FINDER was enabled. So we don't claim that the results generalize to the remaining test targets or any targets for which CULPRIT FINDER was not enabled.

Number of feedback reports: Over the span of the study, we received 377 feedback reports. This is a small fraction of the notification emails sent to developers. Additionally, we received more feedback for some test targets than others. Therefore, we don't claim that received feedback are a good representation of all results.

Confirmation of feedback reports: The feedback reports were manually reported by developers and are not confirmed, i.e. we didn't do any validation on the culprit CLs and assumed they are correctly reported.

Control vs experiment groups: We used only 2 test targets and 2 developers in our control group. So we don't claim that the results of control vs experiment groups generalize to all test targets or all feedback received from developers.

Potential multiple culprits in changelog: In our case studies, we assumed there is a single culprit CL for a test failure and used the first feedback report if more than one are submitted for the same failure. However, this may not be true, i.e. there can be multiple culprit CLs in the changelog.

CULPRIT FINDER runtime: In our feedback reports, we did not automatically record the total time it takes for CULPRIT FINDER to run. Therefore we did not do a systematic study on runtimes. The runtime data provided in Table VI are obtained manually after the fact. As a result, we don't claim any guarantees or generalizations on the runtime.

Flaky Tests: Flaky tests are tests that exhibit both a passing and a failing result with no code or environment changes [11]. In Google, there are tools to prevent CULPRIT FINDER from running for flaky tests. However, in our case studies, we do not know if/which developers used this tool and/or if any of the reported feedback is for flaky test failures.

VIII. RELATED WORK

There are several research areas that are relevant, but not directly comparable, to the work in this paper.

Change impact analysis aims to identify program edits on the codebase and which of those edits might have induced test failures, to help developers during debugging [12], [13]. These techniques focus on finer-grained edits than what is described as a *changelist* in this paper.

Fault localization aids developers to find the root cause of a failing test in source/test code [14]. Existing techniques, such as Tarantula [15], focus on locating faults in the code at a specific version of the system, whereas the work in this paper focuses on identifying the change that introduced the fault in the first place before it can be located in code.

There are also existing techniques that are directly comparable to the work in this paper.

There are machine learning (ML) based models that use information in changes such as author, modified files and size of change. Several 'just-in-time' techniques can predict a change to be *risky* to cause regressions before it is submitted to the repository [16], [17]. Other ML based techniques identify regression introducing changes postsubmit [18], [19]. However, unlike the work in this paper, these ML based techniques require training data, and the ML model needs to be maintained/updated over time.

A recent technique, called Locus, uses information retrieval to match a bug report description to a change using *change hunks*, i.e. code that is modified by a change [5]. Unlike the work in this paper, Locus needs a textual bug report that describes the regression, which requires a developer to manually acknowledge and describe the regression in a bug report, and may not be available immediately.

Finally, the *'git bisect'* tool [8] helps perform a manual binary search in the CL window to help a developer find the culprit CL. This is a manual task, and requires re-running tests.

IX. Conclusion

Identifying changes that introduce regressions to a codebase is critical to keep the momentum on software development, especially in very large scale software repositories with rapid development cycles, such as at Google. Therefore, there is a high demand for automated techniques that can help developers locate the change that introduced a regression while minimizing manual human intervention.

In this paper, we introduce a novel technique to identify changes that induce failures for large/enormous tests, called Culprit Finder. Our technique uses heuristics to automatically filter and rank CLs that might have introduced the regression to help developers identify the root cause.

We evaluate our technique on case studies we ran on 140 projects in Google. Results of our case studies show that each heuristic in our algorithm provides benefit to developers on identifying the change that induces test failures.

Our case studies also suggest some bias in developer behavior, where they report feedback when Culprit Finder is not as successful as they expect.

A. Future Work

Future work consists of augmenting our work with more heuristics, such as logs analysis, where failure logs and stack traces can be compared to the code changes in a CL to check for overlaps. Existing ML based models can also be combined with our technique to improve results.

Another important avenue is testing Culprit Finder on small/medium tests with a more systematic evaluation. Such tests can be re-executed to pinpoint the culprit change and Culprit Finder can be evaluated automatically. There is ongoing work on this at Google.

Finally, another avenue to explore is helping large source code repositories, such as github [20], adopt our algorithm (similar to *git bisect* [8]) and reproduce our results.

Acknowledgment

Authors would like to thank Vivek Ramavajjala for his contributions on enabling us to run our case studies on a large scale by making Culprit Finder easily available for all teams that have large/enormous tests at Google.

Authors would also like to thank Rob Siemborski, Caitlin Sadowski, Ciera Jaspan, John Micco, Atif Memon and Tony Ohmann for valuable discussions and feedback on this paper.

Authors would also like to thank the anonymous reviewers that helped improve this paper with valuable feedback.

References

[1] A. Kumar, "Development at the speed and scale of google," QCon San Francisco, 2010. [Online]. Available: https://www.infoq.com/presentations/Development-at-Google

[2] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, pp. 78–87, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2854146

[3] J. Micco, "Tools for continuous integration at google scale," 2012. [Online]. Available: http://youtube.com/watch?v=KH2_sB1A61A

[4] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[5] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 262–273.

[6] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "edoctor: Automatically diagnosing abnormal battery drain issues on smartphones," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 57–70.

[7] "Google test sizes." [Online]. Available: http://googletesting.blogspot.com/2010/12/test-sizes.html

[8] "git bisect." [Online]. Available: https://git-scm.com/docs/git-bisect

[9] "Blaze." [Online]. Available: http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html

[10] "Bazel." [Online]. Available: http://bazel.io

[11] "Flaky tests at google and how we mitigate them." [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[12] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 432–448.

[13] R. S. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[14] W. E. Wong and V. Debroy, "A Survey of Software Fault Localization," Department of Computer Science, The University of Texas at Dallas, Tech. Rep. UTDCS-45-09, 01 2009.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[16] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[17] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, Aug 2015, pp. 17–26.

[18] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, March 2008.

[19] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90.

[20] "github." [Online]. Available: https://github.com/