Flake Aware Culprit Finding

Tim A. D. Henderson*, Bobby Dorward†, Eric Nickell‡, Collin Johnston§, Avi Kondareddy**
Google LLC
1600 Amphitheatre Pkwy
Mountain View, California, USA 94043
*tadh@google.com, †dorward@google.com, ‡esnickell@google.com §collinj@google.com, **avikr@google.com

***Abstract*—When a change introduces a bug into a large software repository, there is often a delay between when the change is committed and when bug is detected. This is true even when the bug causes an existing test to fail! These delays are caused by resource constraints which prevent the organization from running all of the tests on every change. Due to the delay, a Continuous Integration system needs to locate buggy commits. Locating them is complicated by flaky tests that pass and fail non-deterministically. The flaky tests introduce noise into the CI system requiring costly reruns to determine if a failure was caused by a bad code change or caused by non-deterministic test behavior. This paper presents an algorithm, *Flake Aware Culprit Finding*, that locates buggy commits more accurately than a traditional bisection search. The algorithm is based on Bayesian inference and noisy binary search, utilizing prior information about which changes are most likely to contain the bug. A large scale empirical study was conducted at Google on 13,000+ test breakages. The study evaluates the accuracy and cost of the new algorithm versus a traditional deflaked bisection search.**

I. Introduction

Fast, collaborative, large scale development is enabled by the use of Continuous Integration in a *mono-repository* (monorepo) environment where all of the source code is stored in a single shared repository [1]. In a monorepo, all developers share a single source of truth for the state of the code and most builds are made from "head" (the most recent commit to the repository) using the source code for all libraries and dependencies rather than versioned pre-compiled archives. This enables (among other things) unified versioning, atomic changes, large scale refactorings, code re-use, cross team collaboration, and flexible code ownership boundaries.

Very large scale monorepos (such as those at Google [1], [2], Microsoft [3], and Facebook [4]) require advanced systems for ensuring changes submitted to the repository are properly tested and validated. *Continuous Integration* (CI) [5] is a system and practice of automatically integrating changes into the code repository that serves as the source of truth for the organization. In modern environments, integration involves not only performing the textual merge required to add the change but also verification tasks such as ensuring the software compiles and the automated tests pass both on the new change before it is integrated *and after* it has been incorporated into the main development branch.

$31.00 ©2023 IEEE, DOI TBA

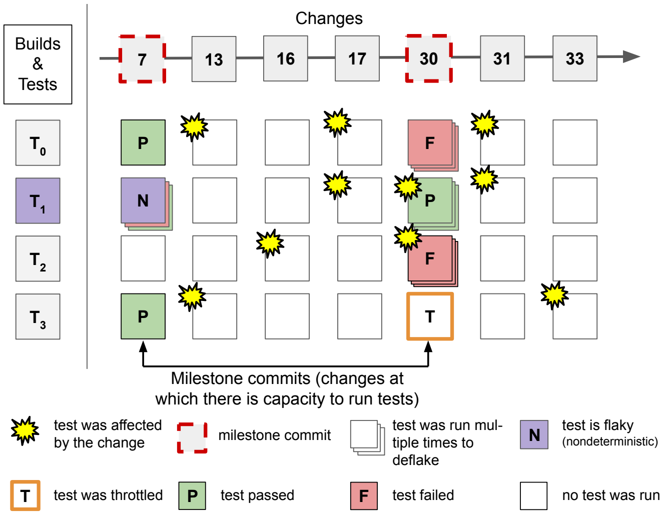

Builds & Tests
Changes
7
13
16
17
30
31
33
T0
T1
T2
T3
P
N
F
P
F
P
T
Milestone commits (changes at which there is capacity to run tests)
test was affected by the change
milestone commit
test was run multiple times to deflake
N test is flaky (nondeterministic)
T test was throttled
P test passed
F test failed
no test was run

Fig. 1: An example timeline showing when tests are run by a continuous integration system. Tests are only run a "milestone" changes where there is backend machine execution capacity to run tests. Some tests are throttled (as the test owner isn't paying for continuous testing), some tests are run multple times to deflake, and finally some tests will not be run because they were not statically affected by the change [6].

Due to resource constraints, tests are only run at specific versions (called *milestones* at Google) and many tests may be skipped because they were not affected by the intervening changes (as determined by static build dependencies) [6]. This process is illustrated in Figure 1. A test is considered passing at a version b if and only if it has a passing result at the most recent preceding affecting version. Formally, if there exists a version a where the test is passing and $a \leq b$ and there does not exist an intervening version c, $a < c \leq b$, which affects the test then the test is passing at version a. A project's status is considered passing at a version b if all tests are passing (using the above definition) at version b.

If a test t is failing (consistently) at a milestone version m, it may not necessarily mean that version m introduced a change which broke test t. As shown in Figure 1 one or more changes between m and the previous milestone at which test t was run may have introduced a breaking change. Figuring out which version introduced the *breakage* is called *culprit finding*.

Many companies now employ automated culprit finding systems. These systems typically implement a (possibly nway) "bisect search" similar to the one built into the Git version control system [7]. Git's bisect command runs a binary search on the versions between the first known breaking version and the last known passing version to identify the version which broke the test.

In the Google environment, there are two problems with a traditional bisection algorithm [7]: (1) it does not account for flaky (non-deterministic) tests, and (2) when using k-reruns to deflake the search accuracy plateaus while build cost continues to increase linearly. **This paper proposes** a new method for culprit finding which is both robust to non-deterministic (flaky) test behavior and can identify the culprit with only logarithmic builds in the best case by using prior information to accelerate the search.

A. Contributions

1) A Flake Aware Culprit Finding (FACF) algorithm.
2) A mathematical model for culprit finding adjusting for non-deterministic test behavior.
3) A large scale empirical study of FACF on Google's mono repository comparing its performance against the traditional bisect algorithm.
4) The study also evaluates the effectiveness: (a) optimizations for FACF, and (b) adding deflaking runs to Bisect.

II. Background

At Google, the Test Automation Platform (TAP) runs most tests that can run on a single machine (and do not use the network except for the loopback interface)[1] and can run in under 15 minutes.[2] Figure 1 illustrates at a high level of how TAP works. First, (not illustrated) tests are grouped into logical projects by the teams that own them. Then, (for accounting and quota allocation purposes) those projects are grouped into very high level groupings called Quota Buckets. Each Quota Bucket is scheduled independently when build resources are available. This means that TAP does not run every test at every version. Instead it only runs tests periodically at select versions called "Milestones" [2], [8], [9].

When scheduling a Milestone, all the tests that are built and run with the same command line flags to the build system[3] are grouped into a set. That set is filtered to remove tests that haven't been affected by a change to themselves or a build level dependency [6], [8], [10]. This is a very coarse grained form of static dependence-based test selection [11], [12], [13]. Google's use of build dependence test selection has been influential and it is now also used at a number of other corporations who have adopted Bazel or similar tools [14], [4]. Finally, that (potentially very large) set of "affected" tests is sent to the Build System [15] which batches them into efficiently sized groups and then builds and runs them when there is capacity.

Since tests are only run periodically even with accurate build level dependence analysis to drive test selection, there are usually multiple changes which may have introduced a fault in the software or a bug in the test. Finding these "bug inducing changes" – which we call **culprits** – is the subject of this paper. The process of finding these changes is aggravated by any non-deterministic behavior of the tests and the build system running them.

A. Flaky Tests

When a test behaves non-deterministically, we refer to it, using the standard term, as a *flaky test*. A flaky test is one that passes or fails non-deterministically with no changes to the code of either the test or the system under test. At Google, we consider all tests to be at least *slightly* flaky as the machines that tests are running on could fail in ways that get classified as a test failure (such as resource exhaustion due to a noisy neighbor resulting in a test exceeding its time limit). We encourage everyone to adopt this conservative view for the purposes of identifying culprits.

An important input to the culprit finding algorithm we are presenting is an estimate of how flaky a test might be, as represented by an estimated probability of the test failing due to a flaky failure. We will denote the estimated probability of failure due to a flake for a test t as $\hat{f}_t$. The actual probability of failure due to a flake will be denoted f_t (it is impossible in practice to know f_t). At Google, we estimate $\hat{f}_t$ based on the historical failure rate of the test. Apple and Facebook recently published on their work in on estimating flake rates [16], [17]. One challenge with flake rate estimation is that a single change to the code can completely change the flakiness characteristics of a test in unpredictable ways. Making the flake rate estimation sensitive to recent changes is an ongoing challenge and one we partially solve in this work by doing live re-estimation of the flake rate during the culprit finding process itself.

B. Culprit Finding

Informally, Culprit Finding is the problem of identifying a version c between two versions a and b such that $a < c \leq b$ where a given test t was passing at a, failing at b, and c is the version which introduced the fault. At Google we consider the passing status of a test to be 100% reliable and therefore version a cannot be a culprit but version b could be the culprit. For simplicity, this paper only considers searching for the culprit for a single test t (which failed at b) but in principle it can be extended to a set of tests T.

Google's version control system, Piper [1], stores the mainline history as a linear list of versions. Each version gets a monotonically, *but not sequentially*, increasing number as its version number called the *Changelist Number* or CL number

[1] In practice the restriction on network usage on TAP is somewhat loose. There are legacy tests that have been tagged to allow them to utilize the network and there is an on going effort to fully eliminate network usage on TAP.

[2] Other CI platforms exist for integration tests that span multiple machines, use large amounts of RAM, or take a very long time to run. The algorithm presented in this paper has also been implemented for one of those CI systems but we will not present an empirical evaluation of its performance in that environment in this paper. In general, Culprit Finding is much more difficult in such environments as tests may use a mixture of code from different versions and network resources may be shared across tests.

[3] An internal version of Bazel https://bazel.build/.

for short. Thus, our implementation and experiments do not consider branches and merges as would be necessary for a Distributed Version Control System (DVCS) such as git. However, only trivial changes to the mathematics are needed to apply our approach to such an environment [7].

Thus, the versions larger than a and smaller or equal to b are an ordered list of items we will call the *search range* and denote as $S = \{s_1, ..., s_{n-1}, s_n\}$ where $s_n = b$. During the search, test executions will be run (and may be run in parallel) at various versions. The unordered set of executions at version s_i will be denoted X_i.

For the purposes of culprit finding, based on the status of failure detected, the build/test execution statuses are mapped to one of 2 possible states: `PASS` or `FAIL`. We denote (for version s_i): the number of statuses that map to `PASS` as $\texttt{PASSES}(X_i)$ and the number of statuses that map to `FAIL` as $\texttt{FAILS}(X_i)$.

C. Bisection

The Bisect algorithm is a specialized form of binary search. It runs tests (or more generally any operation on the code that can return PASS or FAIL) to attempt to identify which commit introduced a bug. Algorithm 1 gives the psuedo-code for this procedure. Bisect is now commonly implemented in most version control systems (see `git bisect` and `hg bisect` as common examples). In the case of Distributed Version Control Systems (DVCS) systems such as 'git' and 'hg' the implementation of bisect is somewhat more complex as they model version history as a directed acyclic graph [7].

The algorithm presented in Alg 1 has two important changes from the standard binary search. First, it takes into account that a test can fail and pass at the same version and the `firstFail` function specifically looks for a failure after the `lastPass`. Second, it takes a parameter k which allows the user to specify that each test execution be run k times. This allows the user to "deflake" the test executions by rerunning failures. Our empirical evaluation (Section V) examines both the cost and effectiveness of this simple modification. While this simple modification to bisect may be reasonably effective for moderately flaky tests, it does not explicitly model the effect of non-deterministic failures on the likelihood of a suspect to be the culprit.

Algorithm 1: Bisect(t, S, k) – pseudo code for a bisect search capable of executing multiple reruns to deflake flaky failures. Can be trivially modified to do an r-ary search by modifying line 17 to divide by r and executing the r tests at suspects: $\{s_m, s_{2 \cdot m}, ..., s_{(r-1) \cdot m}\}$.

```
    Param : $t$, the test to culprit find
    Param : $S$, an ordered list of suspects
    Param : $k$, number of re-runs to do for each test execution
    Result : Culprit as $s_i \in S$ or NONE
 1  let
 2      fn lastPass($X$: set of executions) → int:
 3          if $\exists i \, (PASSES(X_i) > 0)$ :
 4              return max $\left\{ i \;\middle|\; \begin{array}{c} s_i \in S \\ \textbf{and } \texttt{PASSES}(X_i) > 0 \end{array} \right\}$
 5          else:
 6              return 0
 7      fn firstFail($X$: set of executions) → int:
 8          if $\exists i \, (FAILS(X_i) > 0)$ :
 9              return min $\left\{ i \;\middle|\; \begin{array}{c} s_i \in S \\ \textbf{and } \texttt{FAILS}(X_i) > 0 \\ \textbf{and } i > \text{lastPass}(X) \end{array} \right\}$
10          else:
11              return length($S$) + 1
12      fn complete($X$: set of executions) → boolean:
13          return lastPass($X$)+1 == firstFail($X$)
14  in
15      $X = \emptyset$ // a set of executions
16      while not complete($X$) :
17          m = (lastPass($X$) + firstFail($X$))/2
18          $X = X \cup$ executeTestAtVersion($t$, S[m], k)
19      if $c$ := firstFail($X$); $c >$ len($S$) : return NONE
20      else: return S[c]
```

D. Guessing Games and Noisy Searches

One can view the culprit finding scenario as a kind of guessing game. The questioner guesses the test always fails at a given suspect version. A test execution at that suspect acts as a kind of "responder" or "oracle" which answers. The test execution is an unreliable oracle which may answer FAIL when it should have answered PASS. But, the oracle never answers PASS when it should have answered FAIL – note the asymmetry.

This kind of guessing game is a Rényi-Ulam game, which has been well characterized by the community [18]. The main differences from the "classical" games presented by Rényi and separately by Ulam is (a) the format of the questions and (b) the limitations placed on the oracle responding incorrectly. In a common formulation of the Rényi-Ulam game, the questioner is trying to "guess the number" between (for instance) 1 and 1,000,000. Each question is allowed to be any subset of numbers in the search range. The responder or oracle is allowed to lie at most a fixed number of times (for instance once or twice). This formulation has allowed the community to find analytic solutions to determine the number of questions in the optimal strategy. We refer the reader to the survey from Pelc for an overview of the field and its connection with error correcting codes [18]. The culprit finding scenario is "noisy binary search" scenario where the oracle only lies for one type of response. Rivest *et al.* [19] provide a theoretical treatment of this "half-lie" variant (as well as the usual variation).

E. Bayesian Binary Search

Ben Or and Hassidim [20] noted the connection between these "Noisy Binary Search" problems and Bayesian inference. Our work builds off of their formulation and applies it to the culprit finding setting. In the Bayesian setting, consider the probability each suspect is the culprit and the probability there is no culprit (i.e. the original failure was caused by a flake). These probabilities together form a distribution:

$$(\Pr[\text{there is no culprit}]) + \sum_{i=1}^{n} \Pr[s_i \text{ is the culprit}] = 1$$

We represent this distribution with $\Pr[C_i]$ with $i = 1..(n+1)$ where

$$\begin{aligned}\Pr[C_i] &= \Pr[s_i \text{ is the culprit}], \ \forall\, i \in [1,n] \\ \Pr[C_{n+1}] &= \Pr[\text{there is no culprit}]\end{aligned} \tag{1}$$

Initially when the original failure is first detected at s_n and the search range is created, the probability distribution is uniform: $\Pr[C_i] = {}^{1}\!/_{(n+1)}$ for all $i \in [1, n+1]$. Now consider new evidence E arriving. In the Bayesian model, this would *update* our probability given this new evidence:

$$\Pr[C_i|E] = \frac{\Pr[E|C_i] \cdot \Pr[C_i]}{\Pr[E]} \tag{2}$$

In the above equation, the $\Pr[C_i]$ is the *prior* probability and the probability $\Pr[C_i|E]$ is the *posterior* probability. In the culprit finding context, the new evidence is typically newly observed test executions. However, there is no reason we cannot take into account other information as well.

In the Bayesian framework, new evidence is applied iteratively. In each iteration, the prior probability is the posterior probability of the previous iteration. Since multiplication is commutative, multiple pieces of evidence can be applied in any order to arrive at the same posterior probability:

$$\begin{aligned}\Pr[C_i|E_1,E_2] &= \frac{\Pr[E_2|C_i]}{\Pr[E_2]}\frac{\Pr[E_1|C_i]}{\Pr[E_1]}\Pr[Ci] \\ &= \frac{\Pr[E_1|C_i]}{\Pr[E_1]}\frac{\Pr[E_2|C_i]}{\Pr[E_2]}\Pr[Ci]\end{aligned} \tag{3}$$

After the current posterior probability has been calculated, it is used to select the next suspect to execute a test at. In the traditional Bayes approach, the suspect to examine is the one with the maximum posterior probability. However, in the context of a noisy binary search, the suspects are ordered so a PASS at suspect s_i indicates a PASS at all suspects prior to s_i. Following Ben Or and Hassidim [20], we convert the posterior distribution to a cumulative probability distribution and select the first suspect with a cumulative probability ≥ 0.5.

III. Flake Aware Culprit Finding

Now we introduce our algorithm, Flake Aware Culprit Finding (FACF), in the Bayesian framework of Ben Or and Hassidim [20]. As above, let $\Pr[C_i]$ be the probability that suspect s_i is the culprit and $\Pr[C_{n+1}]$ be the probability there is no culprit. These probabilities are initialized to some suitable initial distribution such as the uniform distribution.

To update the distribution with new evidence E in the form of a pass or fail for some suspect k, apply Bayes rule (Equation 2). Note that $\Pr[E]$ can be rewritten by Bayes rule in terms of a sum by marginalizing over the likelihood that each suspect j could be the culprit:

$$\Pr[E] = \sum_{j=1}^{n} \Pr[E|C_j]\Pr[C_j] \tag{4}$$

There are two types of evidence: test executions with state `PASS` or `FAIL` at suspect i. We indicate these as the events P_i and F_i respectively. The marginalized probability of a `PASS` at i given there is a culprit at j is:

$$\Pr[P_i|C_j] = \begin{cases} 1-\hat{f}_t & \text{if } i < j \\ 0 & \text{otherwise}\end{cases} \tag{5}$$

The marginalized probability of a `FAIL` at i given there is a culprit at j is:

$$\Pr[F_i|C_j] = \begin{cases} \hat{f}_t & \text{if } i < j \\ 1 & \text{otherwise}\end{cases} \tag{6}$$

With Equation 5, we write a Bayesian update for `PASS` at suspect k as the following. Given

$$\Pr[C_i|P_k] = \frac{\Pr[P_k|C_i]\Pr[C_i]}{\Pr[P_k]} \tag{7}$$

let

$$\begin{aligned}\Pr[P_k] &= \sum_{j=1}^{n}\Pr[P_k|C_j]\Pr[C_j] \\ &= \sum_{j=k+1}^{n}(1-\hat{f}_t)\Pr[C_j]\end{aligned} \tag{8}$$

then

$$\Pr[C_i|P_k] = \begin{cases} {}^{(1-\hat{f}_t)\Pr[C_i]}\!/_{\Pr[P_k]} & \text{if } k < i \\ 0 & \text{otherwise}\end{cases} \tag{9}$$

The update using Equation 6 for `FAIL` at suspect k is similar. Given

$$\Pr[C_i|F_k] = \frac{\Pr[F_k|C_i]\Pr[C_i]}{\sum_{j=1}^{n}\Pr[F_k|C_j]\Pr[C_j]} \tag{10}$$

let

$$\begin{aligned}\Pr[F_k] &= \sum_{j=1}^{n}\Pr[F_k|C_j]\Pr[C_j] \\ &= \sum_{j=1}^{k}\Pr[C_j] + \sum_{j=k+1}^{n}\hat{f}_t\Pr[C_j]\end{aligned} \tag{11}$$

then

$$\Pr[C_i|F_k] = \begin{cases} {}^{\hat{f}_t\Pr[C_i]}\!/_{\Pr[F_k]} & \text{if } k < i \\ {}^{\Pr[C_i]}\!/_{\Pr[F_k]} & \text{otherwise}\end{cases} \tag{12}$$

A. The FACF Algorithm

Algorithm 2 presents the pseudo code for a culprit finding algorithm based on the above Bayesian formulation. It takes a prior distribution (the uniform distribution can be used), an estimated flake rate $\hat{f}_t$, and the suspects. It returns the most likely culprit or `NONE` (in the case that it is most likely there is no culprit). The FACF algorithm builds a set of evidence X containing the observed passes and failures so far. At each step, that evidence is converted to a distribution using the math presented above. Then, FACF calls Algorithm 3 `NextRuns` to compute the next suspect to run.

`NextRuns` converts the probability distribution into a cumulative distribution. It then selects the suspects with cumulative

Algorithm 2: FlakeAware(t, S, priorDistribution, $\hat{f}_t$) – psuedo code for the FACF algorithm. As with the the Bisect algorithm you can trivially modify it to conduct an r-ary search by passing an r value into "NextRuns" on line 31 and running all all returned indices. Note, the priorDistribution should be of length(S) $+1$ as, by convention, the last element of the distribution is the probability there is no culprit.

```
Param : t, the test to culprit find
Param : S, an ordered list of suspects
Param : priorDistribution, float[] a probability distribution of
        size length(S)+1.
Param : $\hat{f}_t$, an estimated flake rate for test t
Result : Culprit as $s_i \in S$ or NONE

1  let
2      fn passUpdate(priorCulpritPr: float[], k int) → float[]:
3          C = priorCulpritPr
4          C′ = new float[length(C)]
5          $\Pr[P_k] = \sum_{j=k+1}^{n}(1-\hat{f}_t)\Pr[C_j]$
6          for $C_i$ in C :
7              if k < i :
8                  $C'_i = (1-\hat{f}_t)C_i/\Pr[P_k]$
9              else:
10                 $C'_i = 0$
11         return C′
12     fn failUpdate(priorCulpritPr: float[], k int) → float[]:
13         C = priorCulpritPr
14         C′ = new float[length(C)]
15         $\Pr[F_k] = \sum_{j=1}^{k} C_j + \sum_{j=k+1}^{n} C_j\hat{f}_t$
16         for $C_i$ in C :
17             if k < i :
18                 $C'_i = \hat{f}_t C_i/\Pr[F_k]$
19             else:
20                 $C'_i = C_i/\Pr[F_k]$
21         return C′
22     fn asDistribution(X: set of executions) → float[]:
23         dist = copy(priorDistribution)
24         for $S_i$ in S :
25             for j = 0; j < PASSES($X_i$); j++ :
26                 dist = passUpdate(dist, i)
27             for j = 0; j < FAILS($X_i$); j++ :
28                 dist = failUpdate(dist, i)
29 in
30     X = ∅                       // a set of executions
31     dist = asDistribution(X)
32     while max(dist) ≤ threshold :
33         runs = NextRuns(dist, 1)
34         if length(runs) == 0 :
35             break
36         m = runs[0]
37         X = X ∪ executeTestAtVersion(t, S[m], k)
38         dist = asDistribution(X)
39     if c := argmax(dist); c > len(S) : return NONE
40     else: return S[c]
```

Algorithm 3: NextRuns(distribution, k) – psuedo code for a function that computes the next suspects to execute tests on for a given distribution giving the likelihood that each suspect is the culprit. May return no runs if only the last entry in the distribution is above the threshold to run at (indicating there is no culprit). By convention the last element of the distribution is the probability there is no culprit.

```
Param : distribution, a finite, discrete probability distribution
Param : k, number of parallel runs to conduct
Result : The indices of the suspects to execute the next run
         at.

1  cdf = cumulativeDistribution(dist)
2  thresholds = $\{i/(k+1) \mid i = 1..k\}$
3  tidx = 0
4  runs = list()
5  for cidx = 0; cidx < length(cdf)-1; cidx++ :
6      if tidx ≥ len(thresholds) :
7          break
8      while (tidx < len(thresholds)
         and cdf[cidx] ≥ thresholds[tidx]) :
9          if no runs yet at cidx-1 and cdf[cidx-1] > 0 :
10             runs.append(cidx-1)
11         else:
12             runs.append(cidx)
13         tidx++
14 return runs
```

probability over a set of thresholds and returns them. The version of `NextRuns` we present here has been generalized to allow for k parallel runs and also contains a minor optimization on lines 10-11. If the suspect s_{i-1} prior to the selected suspect s_i hasn't been selected to run and its cumulative probability is above 0 then the prior suspect s_{i-1} is selected in favor of s_i. This optimization ensures that we look for the passing run preceding the most likely culprit before looking for a failing run at the culprit.

B. Choosing the Thresholds in NextRuns

Another optimization can be made to NextRuns by changing the thresholds used to select suspects. Instead of simply dividing the probability space up evenly it uses the observation that the information gain from a PASS is more than the gain from a FAIL. Modeling that as $I(P_i) = 1$ and $I(F_i) = 1 - \hat{f}_t$, we can compute the expected information gain for new evidence at suspect s_i as $\mathrm{E}[I(s_i)] = I(P_i)P_i + I(F_i)F_i$. Expanding from the marginalized equations 5 and 6 gives

$$\mathrm{E}[I(s_i)] = (1 - \hat{f}_t)(n + i\hat{f}_t - 1) \tag{13}$$

Computing the cumulative expectation over $\{\mathrm{E}[I(s_1)], ..., \mathrm{E}[I(s_{n+1})]\}$ and normalizing allows us to select the suspect that maximizes the information gain. The empirical evaluation (Section V) examines the effect of this optimization. The importance of the supplied flake rate estimate ($\hat{f}_t$) is also examined in Section V.

C. Prior Probabilities for FACF

The prior distribution supplied to Algorithm 2 does not have to be a uniform distribution. Instead, it can be informed by any other data the culprit finding system might find useful. At Google, we are exploring many such sources of data, but so far are only using two primary sources in production.

First, we have always used the static build dependence graph to filter out non-affecting suspects before culprit finding [6]. In FACF, we take that a step further and, using a stale snapshot for scalability, we compute the minimum distance in the build graph between the files changed in each suspect and the test being culprit-found [2]. Using these distances, we form a probability distribution where the suspects with files closer to the test have a higher probability of being the culprit [21], [22]. Second, we look at the historical likelihood of a true breakage (versus a flaky one) to determine the probability that there is no culprit. Combining that distribution with the one from the minimum build distance forms a prior distribution we use for culprit finding.

Using this scheme, flaky tests tend to get reruns at the s_n (the suspect where the failure was detected) as their first test executions during culprit finding. This is intentional as a large fraction of the search ranges we culprit find tend to be flaky and this reduces the cost of proving that. In the empirical evaluation (Section V), we examine the effect of the prior distribution on both cost and accuracy.

IV. Culprit Verification for Accuracy Evaluation

The previous sections detailed our culprit finding algorithm (FACF) and the baseline algorithm we compare against (deflaked bisection). In order to evaluate the accuracy of a culprit finding system for a particular dataset, we need to know which commits are the actual culprit commits. If the actual culprit commits are known, then we can compute the usual measures (True Positive Rate, False Positive Rate, Accuracy, Precision, Recall, etc...) [23]. Previous work in culprit finding, test case selection, fault localization and related research areas have often used curated datasets (e.g. Defects4J [24]), bug databases [25], [26], [27], and synthetic benchmarks with injected bugs [28], [29]. But our primary interest is in the accuracy of the algorithms in the Google environment, not the accuracy on open source bug databases, curated datasets, or synthetic bugs.

At Google, we do not have a comprehensive database of developer investigations of every continuous integration test failure – indeed there are far too many failures per day for developers to examine and evaluate every single one. We also didn't want to completely rely on human feedback to tell us when our culprit finding systems have identified the wrong culprit – as had been previously done [22].

Instead, we wanted to be able to verify whether or not an identified culprit for a particular test and search range is correct or not. The verification needs to be automated such that the we can operationally monitor the accuracy of the deployed system and catch any performance regressions in the culprit finder. Continuous monitoring ensures a high quality user experience by alerting our team to accuracy degradations.

A. Culprit Finding Conclusions

A culprit finder may draw one of two conclusions:

→ **FI**: `FLAKE_IDENTIFIED` *the culprit finder determined the status transition was caused by non-deterministic behavior.*

→ **CVI**: `CULPRIT_VERSION_IDENTIFIED(version)` *the culprit finder determined a specific* `version` *was identified as the culprit.*

The verification system will label each conclusion examined with either `CORRECT` or `INCORRECT` from which we can define the standard counts for True and False Positives and Negatives. Accuracy, Precision, and Recall can then be directly computed.

B. When is a Culprit Incorrect?

For ease of discussion, we will use the notation introduced in Section II-B for the search range S with s_0 indicating the previous passing version, s_n indicating the version where the failure was first detected, and s_k indicating the identified culprit. Now, consider the two cases

1. `FLAKE_IDENTIFIED` these are incorrect if no subsequent runs pass at the version where the original failure was detected (no flaky behavior can be reproduced).
2. `CULPRIT_VERSION_IDENTIFIED(version)` these are incorrect if we can prove any of the following:
   a) A pass at s_n indicates the failure was a flake.
   b) A pass at the culprit s_k indicates that the culprit was incorrectly identified.
   c) No passes detected at the version prior to the culprit, s_{k-1}, indicates that the culprit was incorrectly identified.

   Additionally the following conditions indicate the culprit may have been misidentified due to a test dependency on either time or some unknown external factor:
   a) No passes are identified at s_0, indicating the test failure may be time or environment dependent (and the conclusion cannot be trusted).
   b) No pass at s_0 is detected prior (in the time dimension) to a failure at s_n.
   c) No failure at s_n is detected prior (in the time dimension) to a pass at s_0.
   d) No pass at s_{k-1} is detected prior (in the time dimension) to a failure at s_k.
   e) No failure at s_k is detected prior (in the time dimension) to a pass at s_{k-1}.
   f) Verification is not reproducible 17 hours later.[4]

Thus, our approach for determining whether or not a culprit is correct is to do extensive reruns for a particular conclusion.

C. How Many Reruns to Conduct to Verify a Conclusion

A test run x times fails every time with the rate $\hat{f}_t^x$. If we want to achieve confidence level C (ex. $C = .99999999$) of verifier correctness, we can compute the number of reruns to conduct for each check with $x = \left\lceil {}^{\log(1-C)}/_{\log\left(\hat{f}_t\right)} \right\rceil$.

D. Sampling Strategy

While it would be ideal to verify every single culprit for every single test, the expense, in practice, of such an operation is too high. We conduct verification with a very high confidence level ($C = 0.99999999$) and a minimum on the estimated flakiness rate of 0.1 to ensure a minimum of 8 reruns for each check even for targets that have not been historically flaky. With such an expensive configuration, we can only afford the following sampling strategy.

1) Randomly sample at 1% of culprit finding conclusions.

[4] The exact number of hours isn't so important. The main thing is to repeat the verification at a suitable interval from the original attempt.

2) Sample the first conclusion for each unique culprit version.

We tag which sampling strategy was used to sample a particular conclusion allowing us to compute statistics for each sampling frame independently.

E. The Verified Culprits Dataset

Using our production culprit finders based on the FACF algorithm and a verifier implementing the above verification strategy, we have produced an internal dataset with 144,130 verified conclusions in the last 60 days, which we use in Section V to evaluate the performance of the algorithms presented in this paper. Note, because the evaluation of an algorithm is also expensive, we only conduct the evaluation on a subset of the whole dataset.

While we are unable to make the dataset public, we encourage others to use our verification approach on their own culprit finding systems and when possible share these datasets with the external community for use in future studies. Such datasets are not only useful for culprit finding research but also for fault localization, test prioritization, test selection, automatic program repair, and other topics.

V. Empirical Evaluation

We empirically evaluated the behavior of the probabilistic Flake Aware Culprit Finding (FACF) algorithm, compared it to the traditional Bisect algorithm [7], evaluated the effectiveness of adding deflaking runs to Bisect, and evaluated the effectiveness of two optimizations for FACF.

A. Research Questions

The following research questions were considered. All questions were considered with respect to the Google environment (see Section V-E for more details about the dataset used).

RQ1: Which algorithm (FACF or Bisect) was more accurate in identifying culprit commits?

RQ2: What difference do algorithmic tweaks (such as adding deflaking runs to Bisect) make to accuracy.

RQ3: How efficient is each algorithm as measured in number of test executions?

RQ4: What difference do algorithmic tweaks (such as adding deflaking runs to Bisect) make to efficiency.

RQ5: Does flakiness affect culprit finding accuracy and cost?

B. Measuring Culprit Finding Accuracy

To measure the accuracy of the culprit finders we produce a binary variable (“correct”) which is true if the culprit finder identified version matched the culprit in the dataset. Accuracy is then defined as the sample proportion: $\textbf{ACC} = {}^{\#\ \text{correct}}/_{\text{Total}}$. Culprit finder “A” is judged more accurate than culprit finder “B” if A’s **ACC** value is higher than B’s and a statistical test (described in Section V-D) determines the difference is significant.

C. Measuring Culprit Finding Cost

We measure cost as the number of test executions performed by the algorithm. Culprit finder “A” is judged lower cost than culprit finder “B” if the mean number of test executions is less than “B” and a statistical test (described in Section V-D) determines the difference is significant.

D. On Our Usage of Statistical Tests

For answering RQ1 and RQ2, we observe the distribution per culprit-finding algorithm of whether or not the algorithm identified the culprit (0 or 1). For answering RQ3 and RQ4, we observe the distribution per culprit-finding algorithm of the number of test executions required to complete culprit finding.

We follow the recommendations of McCrum-Gardner [30] as well as Walpole *et al.* [31] as to the proper test statistics for our analysis. Since data points for a specific culprit range have a correspondence across algorithms, we perform paired statistical tests when feasible. The analysis is complicated by the observation that neither the dichotomous distribution of correctness nor the numerical distribution of number of test executions are normally distributed. Following McCrum-Gardner we opt to use non-parametric tests. Additionally different tests are used for accuracy (dichotomous nominal variable) and number of executions (numerical variable) respectively.

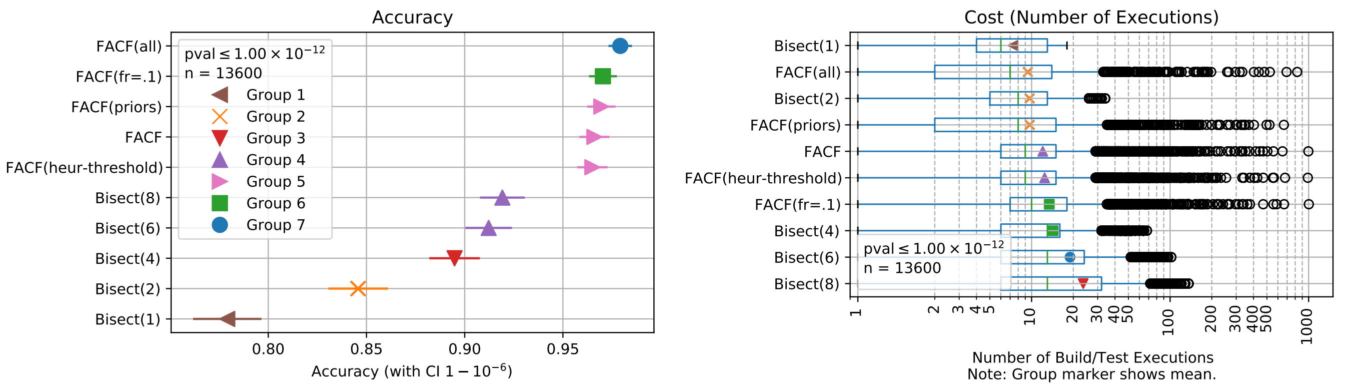
As we have more than 2 groups to consider (10 algorithms), we initially perform omnibus tests over all algorithms to justify the existence of statistical significance over all algorithms. In response to the multiple comparisons problem [31], we compute the corrected significance bound using the formula for the family-wise error rate. Our experimental design has us first conduct the ANOVA significance test to determine if any of the algorithms differed in performance. Then we conduct post-hoc testing to determine which algorithm’s performance differed. For each considered question we conduct $\binom{10}{2} + 1$ significance tests. We have 6 considered questions which we use significance testing to answer, giving us a total $r = 6\left(\binom{10}{2} + 1\right) = 276$ tests.

Solving for the α in the experiment-wise error rate formula from Walpole (pg. 529 [31]) with p standing for the probability of a false rejection of at least one hypothesis in the experiment gives: $\alpha = 1 - (1-p)^{\frac{1}{r}}$. Given $r = 276$ tests and an intended experiment-wise error rate of $p = 0.001$, we conservatively arrive at $\alpha = 10^{-6}$ after rounding down to control for family-wise error rate. We use α as our significance level in all tests.

Cochran’s Q Test is used for significance testing of the accuracy distributions and Friedman’s χ^2 Test for number of executions. Post-hoc testing was conducted with McNemar’s test and Wilcoxon Signed Rank Test respectively.

E. Evaluation Dataset

For the study we used a subset of the dataset produced by the production Culprit Verifier (described in Section IV). The evaluation dataset consists of 13600 test breakages with their required build flags, search range of versions, and the verified culprit. The verified culprit may be an indicator that there was no culprit and detected transition was caused by a



(a) Accuracy, the sample proportion of culprits correctly found, plotted along with a confidence interval. Matching colored markers indicate algorithms which were not significantly different from each other.

(b) Cost, in terms of test/build executions, plotted as a box plot. The plots are sorted by the mean. The mean is indicated by the colored marker. Algorithms with matching markers are not significantly different from one another.

Fig. 2: Accuracy (left) and Cost (right) of the culprit finding algorithms (see Tables I and II for a description of each algorithmic variant). For each figure a group wide significance test was first done (p-value in the legend along with number, n, of matched search ranges in the dataset) and then pair significance tests were conducted to determine which had similar algorithmic performance. Similarly performing algorithms were grouped together into disjoint sets where all algorithms in the group had non-significant pair-wise differences.

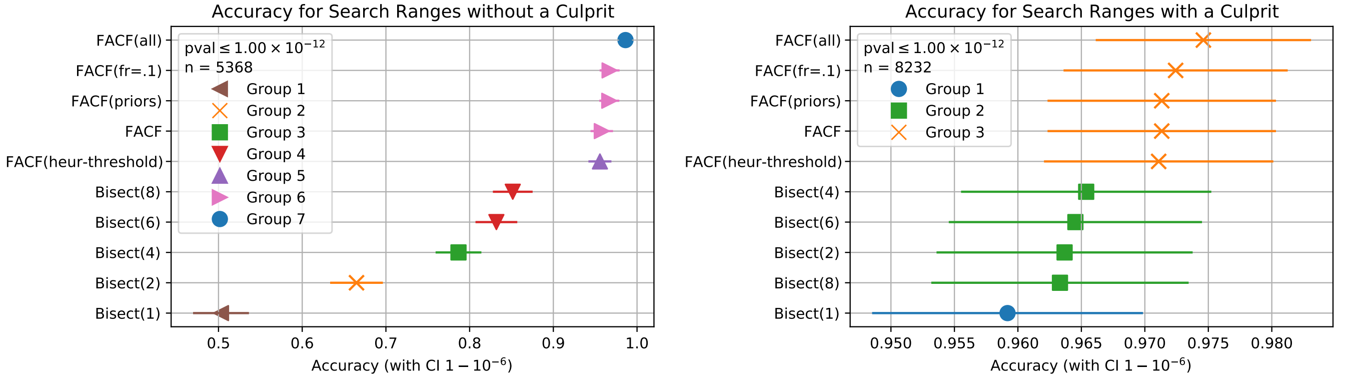


Fig. 3: Accuracy for Search Ranges without Culprits (aka. Flaky Search Ranges) versus Search Ranges with Culprits.

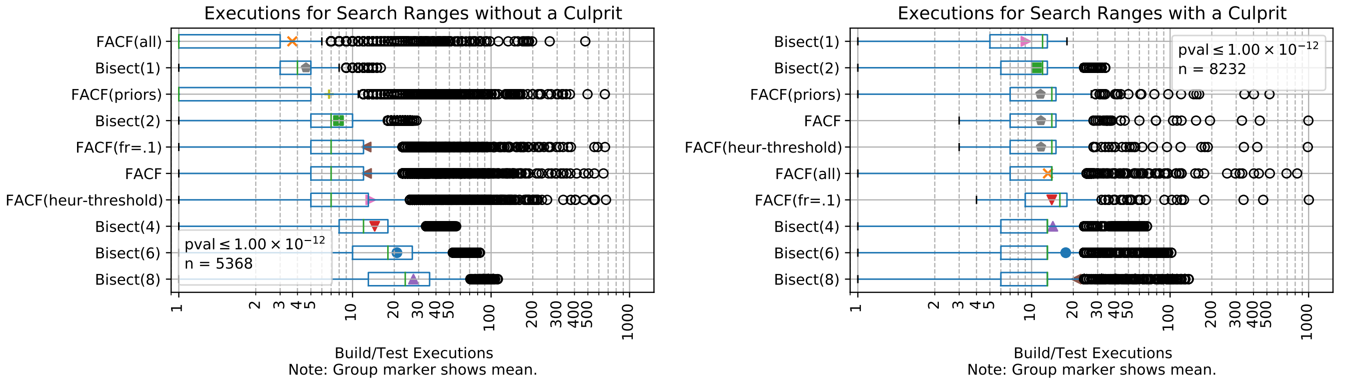


Fig. 4: Cost in test executions for Search Ranges without Culprits (aka. Flaky Search Ranges) versus Search Ranges with Culprits.

flake. Approximately 60% of the items had a culprit version and 40% did not (as these were caused by a flaky failure).

Multiple tests may be broken by the same version. When looking at unique search ranges, we actually see approximately twice as many flaky ranges as non-flaky ones. This hints towards having tighter bounds on which ranges are worth culprit finding as all of the runs for two-thirds of ranges would have been better utilized on more deflaking runs if these two categories are accurately discernable.

TABLE I: VARIANTS OF THE BISECT ALGORITHM STUDIED IN THE EMPIRICAL EVALUATION.

Bisect Variant	Description
Bisect(1)	Bisect (Alg. 1) with 1 test execution per version
Bisect(2)	Bisect (Alg. 1) with 2 test execution per version
Bisect(4)	Bisect (Alg. 1) with 4 test execution per version
Bisect(6)	Bisect (Alg. 1) with 6 test execution per version
Bisect(8)	Bisect (Alg. 1) with 8 test execution per version

TABLE II: VARIANTS OF THE FACF ALGORITHM STUDIED IN THE EMPIRICAL EVALUATION.

FACF Variant	Description
FACF(all)	FACF variant with all improvements enabled
FACF	Basic FACF algorithm (see Alg. 2)
FACF(fr=.1)	FACF algorithm with a minimum flake rate of 11%
FACF(heur-threshold)	FACF using the heuristic thresholds from Section III-B in Alg. 3
FACF(priors)	FACF using a non-uniform prior probability (see Sec III-C)

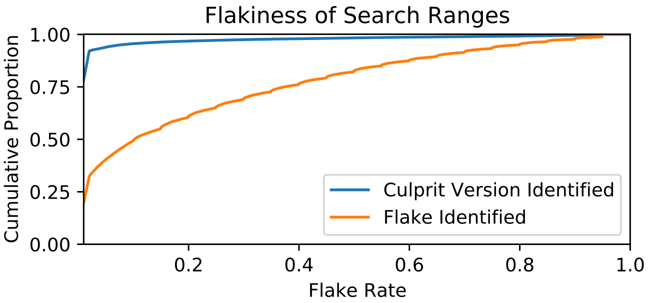


Fig. 5: Cumulative Proportion (y-axis) of Search Ranges with at least one Test under the given Flake Rate (x-axis).

We therefore examine the overall dataset for discrepancies between the two categories. For example, flaky ranges have, on average, 8 tests failing while non-flaky ranges have 224. For the objective of finding unique real culprits, we are interested in the minimum flake rate over targets in a range for flaky and non-flaky ranges, respectively. Fig 5 shows us that, in fact, 99% of real culprit ranges have targets below a flake rate of 75%.

F. Results

1) RQ1: Which algorithm (FACF or Bisect) was more accurate in identifying culprit commits?: All versions of FACF were more accurate overall than all versions of Bisect (Figure 2a). The difference was significant with a p-value $< 10^{-12}$, below the adjusted 10^{-6} significance level required.

2) RQ2: What difference do algorithmic tweaks (such as adding deflaking runs to Bisect) make to accuracy?: The production version of FACF, FACF(all), was most accurate and significantly different than all other algorithms (Figure 2a). This was followed by FACF(fr=.1) which set the floor of 10% on the estimated flake rate for tests. The other FACF variants were equally accurate.

3) RQ3: How efficient is each algorithm as measured in number of test executions?: Figure 2b shows the most efficient algorithm was unsurprisingly Bisect(1) which does not attempt to do any extra executions to deflake test failures. The next most efficient algorithm was FACF(all) which was also the most accurate. As can be seen in the boxplot in Figure 2b, the FACF variants tend to have many more extreme outliers in their cost distribution than the Bisect variants. This is because they account for the prior flake rate. An organization may control their cost by not culprit finding tests which are more than (for instance) 50% flaky. No such upper bound was established for this experiment.

4) RQ4: What difference do algorithmic tweaks (such as adding deflaking runs to Bisect) make to efficiency?: First, both FACF(all) and FACF(priors) had less cost than the standard FACF. This indicates that while FACF(priors) does not improve accuracy it does improve efficiency of the algorithm. In particular many more culprits were found with just two builds.

Second, FACF(heur-thresholds) had no significant effect on cost nor did it show any improvement accuracy. We conclude that although the mathematical motivation may be sound, in the Google environment, we cannot validate a benefit to this optimization at this time.

5) RQ5: Does flakiness affect culprit finding accuracy and cost?: Figures 3 and 4 show the accuracy and cost of culprit finding flaky and non-flaky culprits. For search ranges without a culprit, FACF(all) clearly dominates Bisect in accuracy and even has a small advantage in cost over Bisect(1).

When there is a culprit, the FACF variants are all more accurate than the Bisect variants according to the statistical tests. However, the difference is much less pronounced with Bisect(1) accuracy ∼96.1% and FACF(all) accuracy ∼97.5%. In terms of cost, Bisect(1) and Bisect(2) are the most efficient followed by the FACF variants.

We will note, a production culprit finder cannot know a priori if a breakage was caused by a flake. It would need to do deflaking runs to establish that, costing approximately $\left\lceil {}^{\log(p)}/_{\log\left(\hat{f}_t\right)} \right\rceil$ where p is your desired confidence level. For $p = 10^{-4}$ and $\hat{f} = .2$, this amounts to 6 extra runs, which would significantly increase the cost of bisection. However, as noted in Section V-E, prior flakiness rates could be used to prune out a significant portion of flaky ranges (lowering cost) with a minimal hit to accuracy.

VI. RELATED WORK

A. Culprit Finding

Despite its industrial importance, culprit finding has been relatively less studied in the literature in comparison to topics such as Fault Localization [32], [33] (coverage based [34], [35], [36], [28], [29], [37], information retrieval based [38], [39], [40], or slicing based [41], [12], [42]), Bug Prediction [43], [44], [45], Test Selection [46], [47], Test Prioritization [48], [47] and the SZZ algorithm for identifying bug inducing commits in the context of research studies [49], [50], [51], [52]. This is perhaps understandable as the problem only emerges in environments were it becomes to costly to run every test at every code submission.

The idea of using a binary search to locate the culprit is common, but it is not completely clear who invented it first. The Christian Couder of `git bisect` wrote a comprehensive paper on its implementation [7] but indicates there are other similar tools: “So what are the tools used to fight regressions? [...] The only specific tools are test suites and tools similar to `git bisect`.” The first version of the `git bisect` command appears to have been written by Linus Torvalds in 2005.[5] Torvald's commit message simply states "This is useful for doing binary searching for problems." The source code for the mercurial bisect command `hg bisect` states that it was inspired by the git command.[6] The Subversion (SVN) command, `svn bisect`, also cites the git command as inspiration.[7] Therefore, as far as the authors can tell bisection for finding culprits appears to have been invented by Torvalds.

The Couder paper [7] cites a GitHub repository for a program called "BBChop," which is lightly documented and appears not entirely completed by a user enigmatically named Ealdwulf Wuffinga (likely a pseudonym, as that was a name of the King of East Anglia from 664-713).[8] This program claims to have also implemented a Bayesian search for culprits. We gladly cede the throne for first invention of applying Bayes rule to culprit finding to Ealdwulf and note that our invention was independent of the work in BBChop.

More recently, work has been emerging on the industrial practice of Culprit Finding. In 2013, Ziftci and Ramavajjala gave a public talk at the Google Test Automation Conference in on Culprit Finding at Google [21]. They note, that by 2013 Google has already been using m-ary bisection based culprit finding for "small" and "medium" tests. They then give a preview of an approach elaborated on in Ziftci and Reardon's 2017 paper [22] for culprit finding integration tests where test executions take a long time to run. The authors propose a suspiciousness metric based on the minimum build dependence distance and conduct a case study on the efficacy of using such a metric to surface potential culprits to developers. We utilize a similar build dependence distance as input to construct a prior probability (see Section III-C) in this work but do not directly surface the metric to our end users. Finally, for ground truth they rely on the developers to report what the culprit version is. In contrast, the empirical evaluation we present in this new work is fully automated and does not rely on developers.

In 2017, Saha and Gligoric [53] proposed accelerating the traditional bisect algorithm via Test Selection techniques. They use Coverage Based Test Selection [54], [55] to choose which commits are most likely to affect the test results at the failing commit. This technique can be viewed as fine grained, dynamic version of the static build dependence selection strategy we employ at Google.

In 2019, Najafi *et al.* [27] looked at a similar problem to the traditional culprit finding problem we examine here. In their scenario, there is a submission queue that queues up commits to integrate into the main development branch. It batches these waiting commits and tests them together. If a test fails, they need to culprit find to determine the culprit commit. The authors evaluate using a ML-driven risk based batching approach. In 2022, the same research group re-evaluated the 2019 paper on an open source dataset in Beheshtian *et al.*'s 2022 paper [56]. They found that in the new environment the risk based batching didn't do as well as a new and improved combinatorics based batching scheme for identifying the culprit commits.

James Keenan gave a presentation in 2019 at the Perl Conference in which he presented a concept, *multisection*, for dealing with multiple bugs in the same search range [57]. Ideally, industrial culprit finders would automatically detect that there are multiple distinct bug inducing commits.

In 2021, two papers looked at using coverage based data for accelerating bug localization. An and Yoo [58] used coverage data to accelerate both bisection and SZZ [49] and found significant speedups. Wen *et al.* [59] combined traditional Coverage Based Statistical Fault Localization [34] with historical information to create a new suspiciousness score [60]. This fault localization method could be used in a culprit finding context by using the line rankings to rank the commits.

Finally, in 2022 Frolin Ocariza [61] published a paper on bisecting performance regressions. There are strong connections between a flaky test and a performance regression as a performance regression may not happen 100% of the time. Furthermore, the "baseline" version may randomly have poor performance due to environmental or other factors. Ocariza also arrives at a probabilistic approach, using a Bayesian model to determine whether a run at a particular version is exhibiting the performance regression or not. While mathematically related to our work here and complementary, the approach is distinct. The authors note that their technique can be combined with a Noisy Binary Search citing Karp [62] or with a multisection search citing Keenan [57].

B. Noisy Binary Search

There is a vast literature on Rényi-Ulam games and Noisy Searching problems (doubly so when considering their connection to error correcting codes)! We direct readers to the Pelc survey [18] as a starting point. We will highlight a few articles here. First and foremost, the Ben Or and Hassidim paper [20] from 2008 most clearly explains the problem in terms of a Bayesian inference, which is the formulation we use here. Second, Waeber *et al.* in 2013 [63] discuss the theoretical foundations of these *probabilistic bisection algorithms*. Finally, according to Waeber, Horstein first described this algorithm in the context of error correcting codes in 1963 [64].

VII. Conclusion

Flake Aware Culprit Finding (FACF) is both accurate, efficient, and able to flexibly incorporate prior information on the location of a culprit. A large scale empirical study on real test and build breakages found FACF to be significantly more effective than a traditional bisection search while not increasing cost over a deflaked bisection.

[5] https://github.com/git/git/commit/8b3a1e056f2107deedfdada86046971c9ad7bb87

[6] https://www.mercurial-scm.org/repo/hg/file/52464a20add0/mercurial/hbisect.py, https://www.mercurial-scm.org/repo/hg/rev/a7678cbd7c28

[7] https://metacpan.org/release/INFINOID/App-SVN-Bisect-1.1/source/README

[8] https://github.com/Ealdwulf/bbchop

References

[1] R. Potvin and J. Levenberg, “Why google stores billions of lines of code in a single repository,” *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[2] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming Google-scale continuous testing,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE, May 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[3] K. Herzig and N. Nagappan, “Empirically Detecting False Test Alarms Using Association Rules,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 39–48. [Online]. Available: http://ieeexplore.ieee.org/document/7202948/

[4] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, “Predictive Test Selection,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 91–100. [Online]. Available: https://ieeexplore.ieee.org/document/8804462/

[5] M. Fowler, “Continuous Integration,” 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[6] P. Gupta, M. Ivey, and J. Penix, “Testing at the speed and scale of Google,” 2011. [Online]. Available: http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html

[7] C. Couder, “Fighting regressions with git bisect,” *The Linux Kernel Archives*, vol. 4, no. 5, 2008. [Online]. Available: https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html

[8] J. Micco, “Continuous Integration at Google Scale,” EclipseCon 2013, Mar. 2013. [Online]. Available: https://web.archive.org/web/20140705215747/https://www.eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf

[9] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, “Assessing Transition-Based Test Selection Algorithms at Google,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 101–110. [Online]. Available: https://ieeexplore.ieee.org/document/8804429/

[10] J. Micco, “Tools for Continuous Integration at Google Scale,” Google NYC, Jun. 2012. [Online]. Available: https://youtu.be/KH2_sB1A6lA

[11] D. Binkley, “Using semantic differencing to reduce the cost of regression testing,” in *Proceedings Conference on Software Maintenance 1992*. Orlando, FL, USA: IEEE Comput. Soc. Press, 1992, pp. 41–50. [Online]. Available: http://ieeexplore.ieee.org/document/242560/

[12] S. Horwitz and T. Reps, “The use of program dependence graphs in software engineering,” in *International Conference on Software Engineering*, vol. 9. Springer, 1992, p. 349. [Online]. Available: http://portal.acm.org/citation.cfm?id=24041&amp;dl=

[13] F. Tip, “A survey of program slicing techniques,” *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[14] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A. R. Adl-Tabatabai, “Keeping master green at scale,” *Proceedings of the 14th EuroSys Conference 2019*, 2019.

[15] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall, “Scalable build service system with smart scheduling service,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 452–462. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397371

[16] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, “Modeling and ranking flaky tests at Apple,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. Seoul South Korea: ACM, Jun. 2020, pp. 110–119. [Online]. Available: https://dl.acm.org/doi/10.1145/3377813.3381370

[17] M. Machalica, W. Chmiel, S. Swierc, and R. Sakevych, “Probabilistic Flakiness: How do you test your tests?” Dec. 2020. [Online]. Available: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/

[18] A. Pelc, “Searching games with errors—fifty years of coping with liars,” *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 71–109, Jan. 2002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0304397501003036

[19] R. L. Rivest, A. R. Meyer, and D. J. Kleitman, “Coping with errors in binary search procedures (Preliminary Report),” in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing - STOC ’78*. San Diego, California, United States: ACM Press, 1978, pp. 227–232. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800133.804351

[20] M. Ben-Or and A. Hassidim, “The Bayesian Learner is Optimal for Noisy Binary Search (and Pretty Good for Quantum as Well),” in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. Philadelphia, PA, USA: IEEE, Oct. 2008, pp. 221–230. [Online]. Available: http://ieeexplore.ieee.org/document/4690956/

[21] C. Ziftci and V. Ramavajjala, “Finding Culprits Automatically in Failing Builds - i.e. Who Broke the Build?” Apr. 2013. [Online]. Available: https://www.youtube.com/watch?v=SZLuBYlq3OM

[22] C. Ziftci and J. Reardon, “Who broke the build? Automatically identifying changes that induce test failures in continuous integration at google scale,” *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*, pp. 113–122, 2017.

[23] A. Tharwat, “Classification assessment methods,” *Applied Computing and Informatics*, vol. 17, no. 1, pp. 168–192, Jan. 2021. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1016/j.aci.2018.08.003/full/html

[24] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[25] P. Bhattacharya and I. Neamtiu, “Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging,” in *2010 IEEE International Conference on Software Maintenance*. Timi oara, Romania: IEEE, Sep. 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/5609736/

[26] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, “The Art of Testing Less without Sacrificing Quality,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, May 2015, pp. 483–493. [Online]. Available: http://ieeexplore.ieee.org/document/7194599/

[27] A. Najafi, P. C. Rigby, and W. Shang, “Bisecting commits and modeling commit risk during testing,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, Aug. 2019, pp. 279–289. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338944

[28] T. A. D. Henderson and A. Podgurski, “Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Vasteras: IEEE, Apr. 2018, pp. 93–104. [Online]. Available: https://ieeexplore.ieee.org/document/8367039/

[29] T. A. Henderson, A. Podgurski, and Y. Kucuk, “Evaluating Automatic Fault Localization Using Markov Processes,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Cleveland, OH, USA: IEEE, Sep. 2019, pp. 115–126. [Online]. Available: https://ieeexplore.ieee.org/document/8930843/

[30] E. McCrum-Gardner, “Which is the correct statistical test to use?” *British Journal of Oral and Maxillofacial Surgery*, vol. 46, no. 1, pp. 38–41, Jan. 2008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0266435607004378

[31] R. E. Walpole, Ed., *Probability & Statistics for Engineers & Scientists*, 8th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[32] P. Agarwal and A. P. Agrawal, “Fault-localization Techniques for Software Systems: A Literature Review,” *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–8, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2659118.2659125

[33] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A Survey on Software Fault Localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7390282/

[34] J. Jones, M. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002.

[35] J. A. Jones and M. J. Harrold, “Empirical Evaluation of the Tarantula Automatic Fault-localization Technique,” in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://portal.acm.org/citation.cfm?id=1101949

[36] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, “Extended comprehensive study of association measures for fault localization,” *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, Feb. 2014. [Online]. Available: http://doi.wiley.com/10.1002/smr.1616

[37] Y. Kucuk, T. A. D. Henderson, and A. Podgurski, “Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 649–660. [Online]. Available: https://ieeexplore.ieee.org/document/9402143/

[38] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports,” *Proceedings - International Conference on Software Engineering*, pp. 14–24, 2012.

[39] K. C. Youm, J. Ahn, J. Kim, and E. Lee, “Bug Localization Based on Code Change Histories and Bug Reports,” in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. New Delhi: IEEE, Dec. 2015, pp. 190–197. [Online]. Available: http://ieeexplore.ieee.org/document/7467300/

[40] A. Ciborowska and K. Damevski, “Fast changeset-based bug localization with BERT,” in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 946–957. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510042

[41] A. Podgurski and L. A. Clarke, “A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance,” *IEEE Transactions of Software Engineering*, vol. 16, no. 9, pp. 965–979, Sep. 1990. [Online]. Available: http://dx.doi.org/10.1109/32.58784

[42] 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*. Vancouver BC Canada: ACM, Oct. 2004, pp. 432–448. [Online]. Available: https://dl.acm.org/doi/10.1145/1028976.1029012

[43] C. Lewis, Z. Lin, C. Sadowski, and X. Zhu, “Does bug prediction support human developers? findings from a google case study,” in *Proceedings of the . . .*, 2013, pp. 372–381. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486838

[44] K. Punitha and S. Chitra, “Software defect prediction using software metrics - A survey,” in *2013 International Conference on Information Communication and Embedded Systems (ICICES)*. Chennai: IEEE, Feb. 2013, pp. 555–558. [Online]. Available: http://ieeexplore.ieee.org/document/6508369/

[45] H. Osman, M. Ghafari, O. Nierstrasz, and M. Lungu, “An Extensive Analysis of Efficient Bug Prediction Configurations,” in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. Toronto Canada: ACM, Nov. 2017, pp. 107–116. [Online]. Available: https://dl.acm.org/doi/10.1145/3127005.3127017

[46] E. Engström, P. Runeson, and M. Skoglund, “A systematic review on regression test selection techniques,” *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, Jan. 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584909001219

[47] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, “Test case selection and prioritization using machine learning: A systematic literature review,” *Empirical Software Engineering*, vol. 27, no. 2, p. 29, Mar. 2022. [Online]. Available: https://link.springer.com/10.1007/s10664-021-10066-6

[48] H. de S. Campos Junior, M. A. P. Araújo, J. M. N. David, R. Braga, F. Campos, and V. Ströele, “Test Case Prioritization: A Systematic Review and Mapping of the Literature,” in *Proceedings of the 31st Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2017, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/3131151.3131170

[49] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, p. 1, Jul. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1082983.1083147

[50] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm,” *Information and Software Technology*, vol. 99, pp. 164–176, Jul. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584917304275

[51] M. Borg, O. Svensson, K. Berg, and D. Hansson, “SZZ unleashed: An open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. Tallinn, Estonia: ACM Press, 2019, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3340482.3342742

[52] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, “Exploring and exploiting the correlations between bug-inducing and bug-fixing commits,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn Estonia: ACM, Aug. 2019, pp. 326–337. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338962

[53] R. Saha and M. Gligoric, “Selective Bisection Debugging,” in *Fundamental Approaches to Software Engineering*, M. Huisman and J. Rubin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, vol. 10202, pp. 60–77. [Online]. Available: https://link.springer.com/10.1007/978-3-662-54494-5_4

[54] Z. Q. Zhou, “Using coverage information to guide test case selection in Adaptive Random Testing,” *Proceedings - International Computer Software and Applications Conference*, pp. 208–213, 2010.

[55] S. Musa, A. B. M. Sultan, A. A. B. Abd-Ghani, and S. Baharom, “Regression Test Cases selection for Object-Oriented Programs based on Affected Statements,” *International Journal of Software Engineering and Its Applications*, vol. 9, no. 10, pp. 91–108, Oct. 2015.

[56] M. J. Beheshtian, A. H. Bavand, and P. C. Rigby, “Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2784–2801, Aug. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9392370/

[57] J. Keenan, “James E. Keenan - ”Multisection: When Bisection Isn’t Enough to Debug a Problem”,” Jun. 2019. [Online]. Available: https://www.youtube.com/watch?v=05CwdTRt6AM

[58] G. An and S. Yoo, “Reducing the search space of bug inducing commits using failure coverage,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 1459–1462. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3473129

[59] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, “Historical Spectrum Based Fault Localization,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, Nov. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8873606/

[60] S.-F. Sun and A. Podgurski, “Properties of Effective Metrics for Coverage-Based Statistical Fault Localization,” in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2016, pp. 124–134. [Online]. Available: http://ieeexplore.ieee.org/document/7515465/

[61] F. S. Ocariza, “On the Effectiveness of Bisection in Performance Regression Localization,” *Empirical Software Engineering*, vol. 27, no. 4, p. 95, Jul. 2022. [Online]. Available: https://link.springer.com/10.1007/s10664-022-10152-3

[62] R. M. Karp and R. Kleinberg, “Noisy binary search and its applications,” in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’07. USA: Society for Industrial and Applied Mathematics, 2007, pp. 881–890. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1283383.1283478

[63] R. Waeber, P. I. Frazier, and S. G. Henderson, “Bisection Search with Noisy Responses,” *SIAM Journal on Control and Optimization*, vol. 51, no. 3, pp. 2261–2279, Jan. 2013. [Online]. Available: http://epubs.siam.org/doi/10.1137/120861898

[64] M. Horstein, “Sequential transmission using noiseless feedback,” *IEEE Transactions on Information Theory*, vol. 9, no. 3, pp. 136–143, Jul. 1963. [Online]. Available: http://ieeexplore.ieee.org/document/1057832/