December 2022

Identification of Faulty Software Binaries from End-to-End Tests

Shubham Sharma

Varun Puri

Babu Prasad Elumalai

Identification of Faulty Software Binaries from End-to-End Tests

ABSTRACT

Software products are built from a number of interacting binaries that are rolled out asynchronously, at differing schedules and paces. When there is a problem with a software product such as a failure or sub-par performance, identifying the binaries that are the source of a product failure or sub-par performance is difficult with end-to-end tests. Additionally, an apparently failing test can be a false positive that does require any code changes to fix, yet that demands human attention. This disclosure describes techniques to automatically determine, from end-to-end tests of a software product, code binaries that are at the root of product failure or sub-par performance. For each failure signal, a history is maintained of post-triage manual feedback. The history is used to generate a confidence metric, referred to as a cube score, of a new failure being a true positive. Probable false positive test failures can be filtered out, reducing developer effort to address the apparent test failure.

KEYWORDS

- Software testing
- Code binary
- Canary testing
- Performance testing
- Software rollout
- Cube score
- Triage
- Machine learning

BACKGROUND

Software products are built from a number of interacting binaries that are rolled out asynchronously, at differing schedules and paces. When there is a problem with a software product such as a failure or sub-par performance, identifying the binaries (out of the constituent binaries) that are the source of a product failure or sub-par performance is difficult with end-to-end tests.

Currently, an end-to-end test failure is investigated by human developers to triage the failure and to identify the faulty binary. Such an approach is expensive, labor-intensive, and unscalable, especially when a product comprises hundreds or thousands of binaries. Developers usually have in-depth knowledge of binaries that are personally developed or maintained by them; however, it is rare to have developers that have a horizontal understanding across the large number binaries that together comprise a software product. Additionally, in some instances, an apparently failing test can merely be a false positive that does not require code changes to fix, yet that demands human attention. Typically, no history of false positives generated by a given test is maintained.

DESCRIPTION

This disclosure describes techniques that utilize end-to-end tests of a software product to automatically identify the binaries (from a group of constituent binaries) at the root of product failure or sub-par performance. Additionally, the techniques maintain, for each failure signal, a history of manual feedback on post-triage false positivity. The history is used to generate a confidence metric, referred to as a cube score, of a new failure being a true positive. Test failures with a high likelihood of being false positives are filtered out using this metric to reduce

developer effort corresponding to the apparent test failure. The cube score can also be used to more accurately triage test failures.

The techniques use the timestamps of rollout when it is active for the test and the timestamps of test failure to correlate test failure to rollout using event recency, e.g., by building an event chronology. A rollout is active on a certain test when sections of the software newly incorporated into the rollout are visible to the test. Human intervention for triaging and determining the faulty binary is reduced or eliminated.

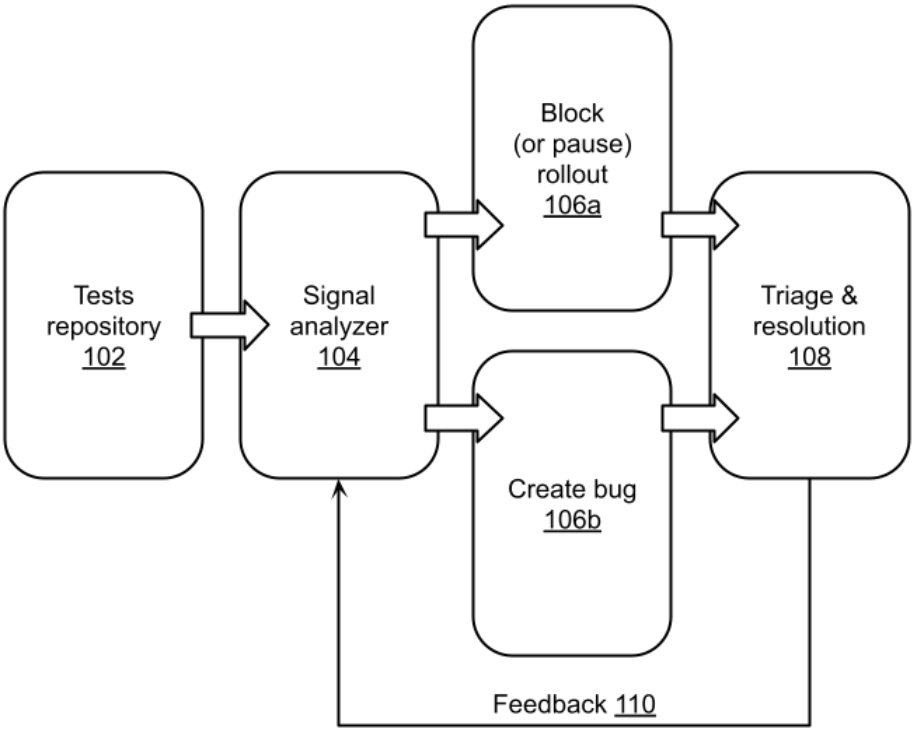

Fig. 1: Accurate identification of faulty software binaries from end-to-end tests

Fig. 1 illustrates techniques for accurate identification of faulty software releases from end-to-end tests, specifically, the identification of one or more binaries from constituent binaries of a software product alongside the confidence metric (cube score) that a failing binary is a true positive. The environment illustrated in Fig. 1 can be situated in a canary-testing region of

software development, e.g., a region where customer-representative, end-to-end tests can be run continuously and new versions released to an initial, relatively small, set of customers.

A test repository (102) includes test suites; test metadata, e.g., histories and captured metrics of past tests; etc. The tests are registered alongside information such as the rollouts that the tests are sensitive to. A signal analyzer (104) analyzes failure signals, e.g., signals relating to test failures; underperformance; unexpected spikes in the load on processor, memory, or other resources; functional deviations; feature defects; etc. The signal analyzer identifies binaries that are potentially faulty and also assigns a confidence to that determination. The signal analyzer is described in greater detail below.

Depending on the severity of the test failure, e.g., confidence in true positivity as determined by the signal analyzer, the software rollout to customers can be paused/ blocked (106a) and/or a bug report created (106b). The test failure is triaged manually and resolved (108). The resolution of the test failure is fed back (110) to the signal analyzer, enabling it to assign a more accurate true positivity confidence metric to a similar pattern of test failures that occur in the future.

Signal analyzer

The signal analyzer identifies binaries that are potentially faulty and also assigns a confidence metric. The sensitivity of end-to-end tests to a group of rollouts is set. Bad rollouts are determined using rollout timestamps from each binary rollout, test-failure timestamps from each end-to-end test, chronology-based heuristics, etc. For example, if an end-to-end test passed for a version built at a time T0 but failed at a version built at a time T1>T0, then one or more binaries introduced between times T0 and T1 can potentially be defective and are marked as

such. However, as explained before, a mere test failure need not mean a defective binary; rather, the test failure can be a false positive.

The signal analyzer assigns a confidence, referred to as a cube score, to a determination that a particular test failure is a true positive. The cube score for a given test signal is dynamically computed based on statistical and historical analysis of the test signal. For example, the feedback provided by the human who triages or resolves an issue, in the form of a false positive or true positive report, is utilized when computing the true positive rating of the test signal. The cube score can be utilized prior to sending a pause or block-rollout signal to improve the accuracy of pausing or blocking a rollout.

```
For the ith data point, the weights[i] will be computed as:
weights[i] = (1/weeksDelta[i])

wherein weeksDelta[i] is the difference in weeks of that run with the current run
rounded to the next largest integer (ceil function)

The weighted average will then be computed as:
weightedAverage = Sum(weights[i] * FalsePositivity[i]) / Sum(weights[i])

wherein FalsePositivity[i] will be 1 if (FalsePositive == True) else 0.

CubeScore for the current test run = (1 - false positivity ratio)
```

Fig. 2: Example pseudo code to compute the cube score

Fig. 2 illustrates an example pseudo code to compute the cube score. As illustrated, past test runs, e.g., over the last n months, can be used to compute the cube score. The false positivity ratio is a weighted average that gives relatively more importance to recent false positives than older ones.

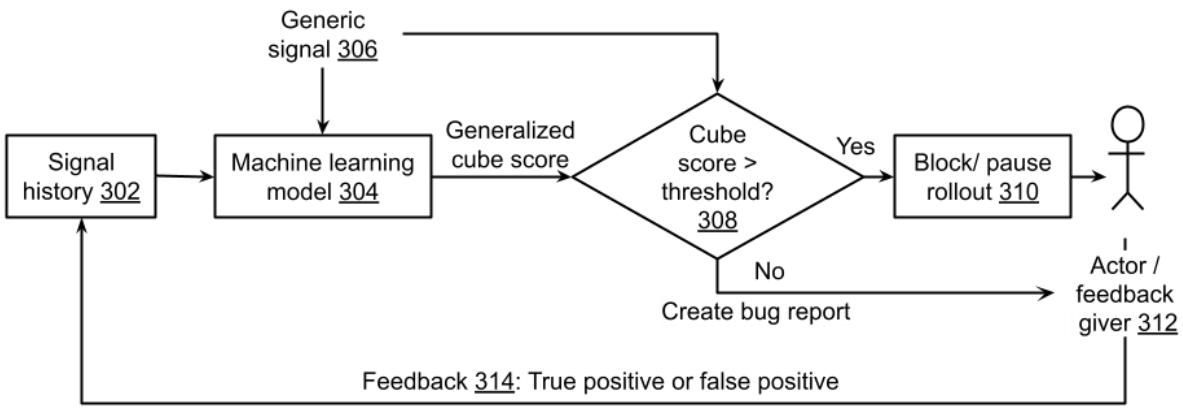

Fig. 3: A generalized cube score

As illustrated in Fig. 3, instead of using a statistical average of historical runs, the cube score can be computed using a machine learning algorithm. Signal history (302) is fed to a machine learning (ML) model (304) that is trained to determine the confidence that a given generic signal (306) is a true positive. If the output generalized cube score from the ML mode exceeds a threshold (308), the rollout is paused or blocked (310), and a gated version of the generic signal is passed on to a human actor (312), who acts on the signal and reports if the signal was indeed a true positive.

If the generalized cube score doesn't exceed a threshold, a bug report is created and passed on to the human actor (312), who acts on the bug report and reports if the signal was indeed a false positive. The report created by the human actor is provided as feedback (314) to be made part of signal history, which in turn can improve the performance of the ML model. In this manner, the cube score can be generalized and used to improve the accuracy of any generic signal that is linked to an increase in manual intervention. Additionally, the knowledge of code paths changed in a rollout can also be fed to the machine learning model to improve the accuracy with which the faulty binary is identified.

The described techniques are applicable generally to software products that are built from multiple interacting binaries. The techniques enable testing and certification of complex software deployments. End-to-end tests can be performed in environments and workloads that closely resemble those of the end customer and can reduce the number of defects that are released into customer environments.

CONCLUSION

This disclosure describes techniques to automatically determine, from end-to-end tests of a software product, code binaries that are at the root of product failure or sub-par performance. For each failure signal, a history is maintained of post-triage manual feedback. The history is used to generate a confidence metric, referred to as a cube score, of a new failure being a true positive. Probable false positive test failures can be filtered out, reducing developer effort to address the apparent test failure.