March 2021

Test Coverage Analysis by Diffing Production Logs Against Integration Test Logs

Yifan Gao

Ben Rafferty

Tylor Sampson

Joseph Kubik

Test Coverage Analysis by Diffing Production Logs Against Integration Test Logs

ABSTRACT

This disclosure describes tools and techniques to analyze input and configuration coverage of software by performing large scale analysis of production and integration test data in a privacy-compliant manner. Test coverage analysis is made possible for highly complex input schema or for input schema that evolve with time. The techniques uncover the input-subspace that has been thus far integration-tested and the input-subspace that remains untested (known as test-gaps). When gathering data on how a specific software input/configuration is used in production, the techniques determine not only if they are used, but also how often they are used. Such data can help a developer evaluate the severity associated with a specific test gap.

KEYWORDS

- Unit testing
- Integration testing
- Blackbox testing
- Whitebox testing
- k-anonymity
- Differential privacy
- Test gap

BACKGROUND

Unit testing of software entails the testing of isolated modules that comprise a software package; integration testing, on the other hand, tests the modules as an integrated entity through the overall API. Unit testing can be measured by using code coverage (or line coverage), where code paths executed by the unit tests are gathered. The methodology to measure integration test coverage, however, is less established. Here we make the following observations:

- While line coverage is easy to gather from unit tests, it is hard to gather from integration tests without instrumenting the software under test (SUT). Specifically, adding instrumentation into production code can bring in security and/or performance risks.
- A developer is interested not only in how software is executed in its implementation but also how software is used through its API. For example, it can be valuable to determine and to analyze integration test gaps by treating the SUT as a blackbox instead of a whitebox.
- When evaluating application programming interface (API) test coverage, it is advantageous to know not only what parameters are made available by the code but also which arguments are actually provided in production. The knowledge of production usage patterns can guide the setup of continuous integration (CI) testing, which reduces the risk of regression. When gathering data on how a specific software input/configuration is used in production, it is advantageous to know not only if they are used, but also how often they are used. Such data can help a developer evaluate the severity associated with a specific test gap. To obtain coverage insights into production usage patterns, production data (which may include user data) can be gathered and analyzed. However, any gathering user data can only occur with specific user permission and user privacy must be protected.

DESCRIPTION

This disclosure describes tools and techniques to obtain test coverage for input and configuration of software by performing large scale analysis of production and test data in a privacy-compliant manner. Test coverage analysis is made possible for highly complex input schema or for input schema that evolve with time. The techniques determine the input-subspace

that has been thus far integration-tested and the input-subspace that is used in production but remains untested (known as test gaps). Privacy protection is provided using differential privacy (or *k*-anonymity), which, although a well-researched and field-proven technique for privacy protection, has thus far not been used for test coverage analysis.

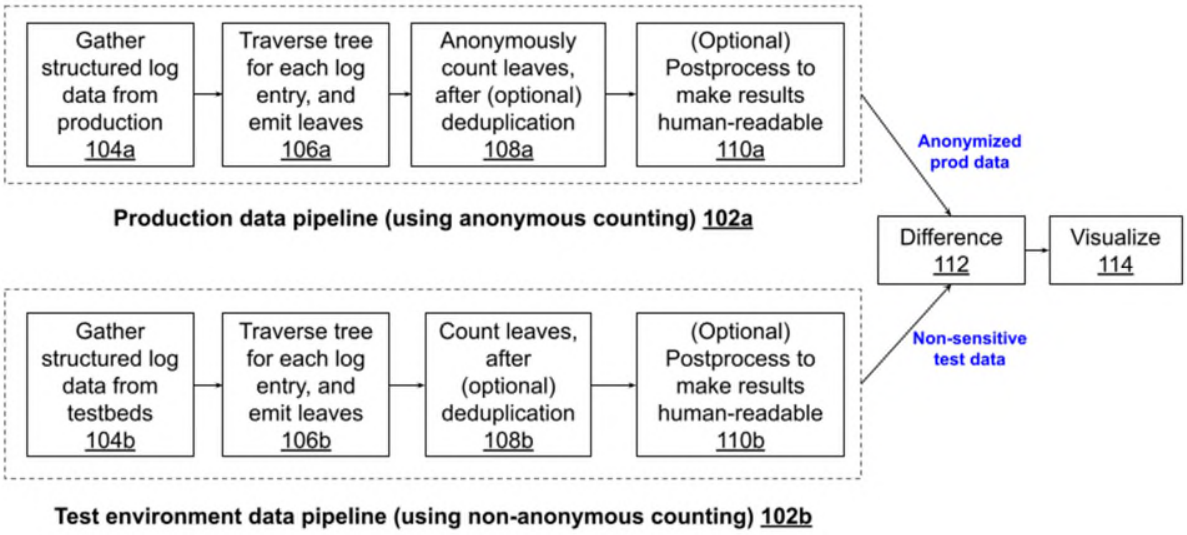

Fig. 1: Integration test coverage analysis by diffing production logs against test logs

Fig. 1 illustrates integration test coverage analysis by diffing production logs against test logs, per the techniques of this disclosure. In an example implementation, the production data has a batch pipeline (102a) and the test environment data has a batch pipeline (102b). The production data pipeline produces pathways (e.g., inputs and configurations) exercised by the set of end-users, in a privacy-compliant manner. The test environment pipeline produces pathways (e.g., inputs and configurations) exercised during testing. The outputs of the two batch data pipelines are diffed (112) and visualized (114) to uncover those regions of input space exercised by the end-user that have thus far been tested and those that have not (test gaps).

An example test gap that can be uncovered by the described tool is a Boolean variable that the customer has exercised to both `true` and `false`, but which the testing procedure has only

exercised to `true`. Not all test gaps are of equal importance. For example, if a field has only a few possible values and only 50% of the possible values have been exercised, then it can be considered as a relatively big production risk. On the other hand, if a field has a very large number of possible values (e.g., it is an integer), then exercising only a few representative values (minimum, maximum, typical) can eliminate or substantially reduce production risk. A risk-scoring system may be built into the visualization in order to reflect the relative importance.

Structured data is gathered from a data logger (production, 104a; testbed, 104b) that records the inputs and configurations consumed by the software under test. The gathered data can be in a serialized, structured format, e.g., XML, JSON, protobuf, etc., and can be used as a group-by aggregator. Examples of data can include the loaded amounts of virtual machine CPU and RAM; the host kernel version; the host hardware type; the number of virtual devices and their configurations; the network configurations, etc.

The structured data is traversed by treating it as a tree and by emitting single leaves (production, 106a; testbed, 106b). A single leaf is defined as one of the following.

- A scalar field, singular or repeated with one element, e.g., a Boolean, an integer, a string, etc.
- Any sub-message in its entirety, if the field path identifier, which describes the path of the sub-message relative to root, is whitelisted or allowed to be treated as leaf per default policy. For example, leveraging the logical locality relationship between `key` and `value` in a map field `{key, value}` with a scalar `value` in a processed protobuf message, the map field is treated as a single leaf.
- A non-value metadata about a field, e.g., the length of a repeated field, the presence or null of an optional field, etc. For such metadata, the emitted value can be a tuple of `{field_path_identifier, metadata}`.

Tree traversal can be carried out using, for example, the reflection capability of protobuf. Each piece of original data is a proto (of protobuf) comprising many single leaves organized in a tree structure that breaks apart after traversal into a vector of single leaves. The tree topology is intentionally discarded; after traversal, there is conceptually no tree. Fig. 2 illustrates an example of tree traversal that results in a vector of single leaves.

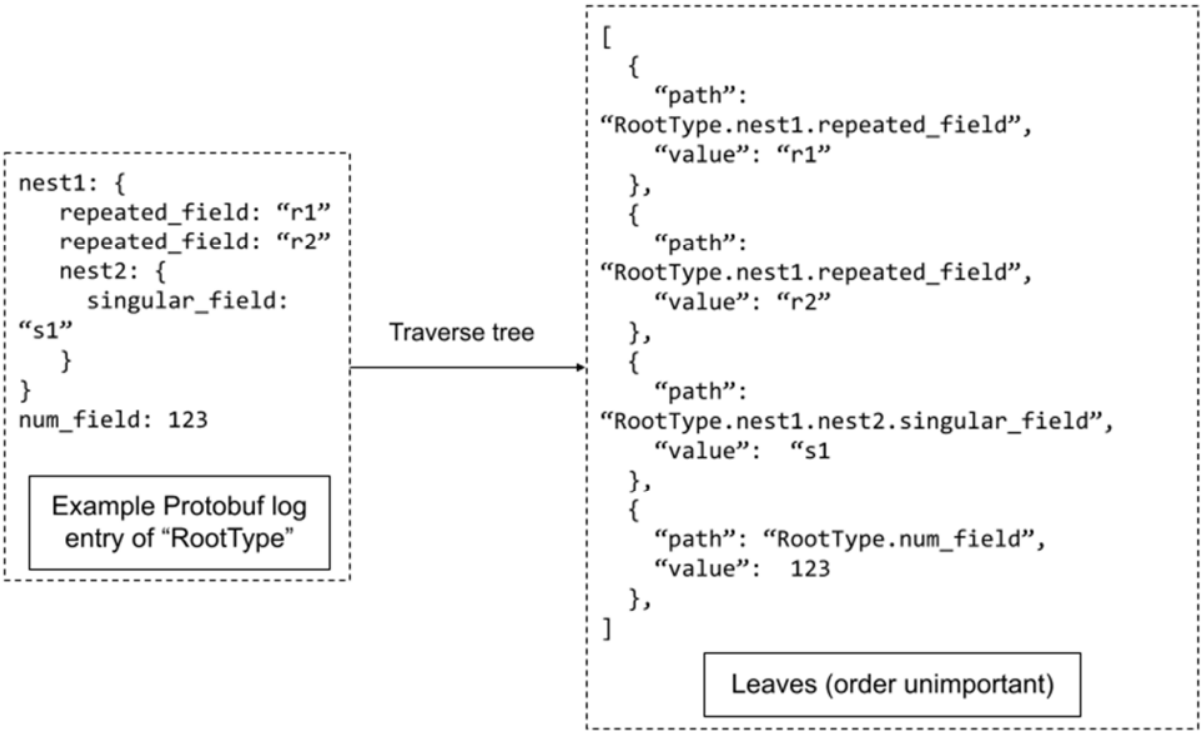

Fig. 2: Traversing a tree to generate single leaves

The leaves, which are regarded as keys, are counted, after deduplication is (optionally) performed (production, 108a; testbed, 108b). For example, in analyzing virtual machine inputs and configurations, deduplication can be performed by keeping the latest entry if the same leaf is logged multiple times by the same VM and/or customer. For the production batch pipeline, privacy is preserved using differential privacy (or *k*-anonymity) aggregation using single leaves as group-by keys and the customer's unique ID as privacy key. For the test environment batch pipeline, non-anonymous aggregation is performed. If the access control and the wipe-out

compliance for storage of aggregated results are implemented separately, production data can also be aggregated in a non-anonymous manner, when specific permissions for such aggregated data are enforced. To avoid the downstream modules that consume data generated by the production and testing pipelines having to repeat tree traversal of structured data, a human-readable string representation of the leaves is generated (production, 110a; testbed, 110b).

The data aggregated as described above is anonymous when differential privacy is used, enabling further analysis such as diffing (112). The diffing can be either spatial or temporal. For example, spatial diffing can include comparing the aggregated data between production and tests to detect field values that are missing a test. Temporal diffing can include monitoring the change or trend of field values in time, thereby detecting unexpected or surprising changes that signal reliability risks. The diffing can be enabled through human-configured diffing policy, machine learning, heuristics, etc.

Reporting front-ends can be used to visualize the data (114) and, in a privacy-compliant manner, automatically create bug reports when new test gaps emerge. For example, a dashboard can consume the diffed log data, perform post-processing (e.g., using SQL), and report a periodic (e.g., weekly) synopsis of test coverage.

Because the data reported by the pipeline is anonymous, an authorized human may need to access the original datastore in order to investigate or to debug. Given the field path and the value reported by the pipeline, a SQL query can assist in debugging. For example, the pipeline can dynamically generate an SQL query, enabling one-click debugging by human engineers. The SQL query itself is anonymous; to run the query, the human engineer is separately authorized into the original datastore, in strict compliance with applicable data access policies.

The ability to perform differential privacy or *k*-anonymity aggregation guarantees full-wipeout compliance without depending on human-configured PII (personally identifiable information) redaction and access control, or human effort and judgement to decide the likely sensitivity of individual fields. Differential privacy also guarantees complete anonymity of the data reported by the tool, enabling downstream modules to consume and reinterpret results where such downstream modules are unable to perform customer data wipeout. For example, test gap reports generated by the described tool are anonymous (wipeout-compliant), thereby enabling automatic bug creation in a downstream bug tracker for test coverage gaps.

The deduplication performed before aggregation makes meaningful the frequency of appearance of each input or configuration with reference to how heavily it is used in production. For example, the logging of a virtual machine (VM) configuration can be triggered when the VM is created, when it is mutated, or periodically even when there are no VM-changing events. Without deduplication, it is hard to tell what the aggregated counts really mean.

The designation of an entire sub-message as a single leaf, per the described techniques, is a flexibility made possible by the principle of aggregating based on leaf instead of scalar. By conveniently marking any sub-message as a single leaf, locality is effectively leveraged: two scalars in the same sub-message are more likely to be logically related, than two scalars distributed far away from each other in a message tree.

In this manner, the described tools and techniques enhance the quality and reliability of software using input and configuration logs collected at production and at testing. As described above, the techniques of differential privacy and/or *k*-anonymity are employed to ensure privacy of user data. The data may be collected by an upstream logging system which ensures that users are provided with indications that certain data may be utilized for maintaining and improving the

services received, and are provided with options to select whether their data can be utilized for such purposes, or to decline permission for use of such data. Therefore, only user-permitted data is used for test gap analysis.

Further to the descriptions above, a user may be provided with controls allowing the user to make an election as to both if and when systems, programs or features described herein may enable collection of user information (e.g., any user data), and if the user is sent content or communications from a server. In addition, certain data may be treated in one or more ways before it is stored or used, so that personally identifiable information is removed. For example, a user's identity and data is treated so that no personally identifiable information can be determined for the user, or a user's geographic location may be generalized where location information is obtained (such as to a city, ZIP code, or state level), so that a particular location of a user cannot be determined. Thus, the user may have control over what information is collected about the user, how that information is used, and what information is provided to the user.

CONCLUSION

This disclosure describes tools and techniques to obtain test coverage for input and configuration of software by performing large-scale analysis of production and test data in a privacy-compliant manner. Test coverage is made possible for highly complex input schema or for input schema that evolve with time. The techniques determine the input-subspace that has been thus far integration-tested and the input-subspace that remains untested (known as test gaps). When gathering data on how a specific software input/configuration is used in production, the techniques determine not only if they are used, but also how often they are used. Such data can help a developer evaluate the severity associated with a specific test gap.

REFERENCES

1. Chambers, Craig, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: easy, efficient data-parallel pipelines." ACM Sigplan Notices 45, no. 6 (2010): 363-375.