Provenance and Pseudo-Provenance for Seeded Automated Test Generation

Alex Groce
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ 86011

Josie Holmes
Pennsylvania State University

Abstract

Many methods for automated test generation, including some that explicitly use machine learning (and some that use ML more broadly conceived) derive new tests from existing tests (often referred to as seeds). Often, the seed tests from which new tests are derived are manually constructed, or at least simpler than the tests that are produced as the final outputs of such test generators. We propose annotation of generated tests with a *provenance* (trail) showing how individual generated tests of interest (especially failing tests) derive from seed tests, and how the population of generated tests relates to the original seed tests. In some cases, post-processing of generated tests can invalidate provenance information, in which case we also propose a method for attempting to construct "pseudo-provenance" describing how the tests *could* have been (partly) generated from seeds.

1 Seeded Automated Test Generation

Automatic generation of software tests, including (security) fuzzing [20, 6], random testing [18, 8, 15], search-based/evolutionary testing [3], and symbolic or concolic execution [5, 4, 2, 16, 12, 22] is essential for improving software security and reliability. Many of these techniques rely on some form of learning, sometimes directly using standard algorithms, and sometimes in a more broadly conceived way. In fact, using Mitchell's classic definition of machine learning as concerning any computer program that improves its performance at some task through experience [14], almost all non-trivial automated test generation algorithms are machine-learning systems, with the following approximate description:

1. Based on results of running all past tests (T), produce a new test $t' = f(T)$ to execute.
2. Execute t' and collect data d' on code coverage, fault detection and other information of interest for the execution of t'.
3. $T = \mathrm{update}(T, t', d')$
4. Go to step 1.

Performance here (in Mitchell's sense) is usually measured by collective code coverage or fault detection of tests in T, or may be defined over only a subset of the tests (those deemed most useful, output as a test suite). The function f varies widely: f may represent random testing with probabilities of actions determined by past executions [1], a genetic-algorithms approach where tests in T are mutated and/or combined with each other, based on their individual performances [13, 3, 20], or an approach using symbolic execution to discover new tests satisfying certain constraints on the execution path [5, 4, 2]. A common feature however, is that many methods do not begin with the "blank slate" of an empty T. Instead, they take as an initial input a population of tests that are thought to be high-quality (and, most importantly, to provide some guidance as to the structure of valid tests), and proceed to generate new tests from these *seed* tests [20, 16, 12, 22, 17]. Seed tests are usually manually generated, or tests selected from a previously generated suite for their high coverage or fault detection [19, 7]. It is generally the case that seed tests are more easily understood by users than newly generated tests.

For example, consider the extremely popular and successful American Fuzzy Lop (AFL) tool for security fuzzing [20]. It usually begins fuzzing (trying to generate inputs that cause crashes indicating potential security vulnerabilities) from a corpus of "good" inputs to a program. When a corpus input is mutated and the result is "interesting," by a code-coverage based heuristic, the new input is added to the corpus of tests to use in creating future tests. Many tools, considered at a high level, operate in the same fashion, with the critical differences arising from engineering aspects (how tests are executed and instrumented), varied heuristics for selecting tests to mutate, and choice of mutation methods. AFL records the origin of each test in its queue in test filenames, which suffices in AFL's case because each test produced is the result of a change to a single, pre-existing test, in most cases, or the merger of two tests, in rarer cases.

This simple approach works for cases where the provenance of a test is always mediated by mutation, making for clear, simple "audit trail." However, a more complex approach is required when the influence of seeds is probabilistic, or a test is composed of (partial) fragments of many tests. Moreover, AFL provides no tools to guide users in making use of what amounts to an internal book-keeping mechanism, rather than an output designed for human examination. Finally, tests, once generated, are frequently manipulated in ways that may make provenance information no longer valid: a test produced from two seed tests (or seed test-derived tests) may be reduced [21] so that one of the tests no longer is present at all, for example.

In this paper, we propose to go beyond the kind of simple mechanisms found in AFL, and offer the following contributions:

- We present an implementation of provenance for an algorithm that involves generating new tests from partial sequences from many seed tests.

- We discuss ways to present information about not just the provenance of a single test, but the impact on future tests of initial seed tests. While single-test provenance is useful for developers debugging, information on general impact of seeds is more important for design and analysis of test generation configuration and algorithms.

- We identify test manipulations that partially or completely destroy/invalidate provenance information, and propose an algorithm for producing a *pseudo-provenance*, showing how the tests generated *could* have been generated from seeds, even if they were not actually thus generated, and discuss abstractions that allow the creation of such a psuedo-provenances in more cases.

2 A Simple Seeded Generation Algorithm with Provenance

We implemented a novel test generation technique for the TSTL [11, 9, 10] test generation language and tool for Python. In this approach, the seed tests are split into short sub-sequences of length k. In place of the usual algorithm for random testing, where a new test action is randomly chosen at each step during testing, our approach always attempts to follow some sub-sequence, in a best-effort fashion (if the next step in the current sub-sequence is not enabled, it is skipped). When a test generated in this fashion covers new code (the usual metric for deciding to learn from a test), it too is broken into sub-sequences and they are added to the sub-sequence pool.

3 Presenting Collective Provenance Information

4 Test Manipulations and Pseudo-Provenance

4.1 A Greedy Pseudo-Provenance Algorithm

Acknowledgments

Use unnumbered third level headings for the acknowledgments. All acknowledgments go at the end of the paper. Do not include acknowledgments in the anonymized submission, only in the final paper.

References

[1] James Andrews, Felix Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Automated Software Engineering*, pages 144–153, 2007.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, pages 209–224, 2008.

[3] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[5] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[6] Peter Goodman. A fuzzer and a symbolic executor walk into a cloud. `https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems/`, August 2016.

[7] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*. accepted for publication.

[8] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[9] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[10] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[11] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.

[12] Paul Dan Marinescu and Cristian Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.

[13] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[14] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[15] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[16] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, 2011.

[17] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2016.

[18] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[19] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[20] Michal Zalewski. american fuzzy lop (2.35b). `http://lcamtuf.coredump.cx/afl/`, November 2014.

[21] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[22] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.