Provenance and Pseudo-Provenance for Seeded Learning-Based 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 software 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 [31, 8], random testing [29, 14, 24], search-based/evolutionary testing [5], and symbolic or concolic execution [7, 6, 2, 25, 21, 33] is essential for improving software security and reliability. Many of these techniques rely on some form of learning, sometimes directly using standard algorithms [20, 26, 3, 15] such as reinforcement learning [12, 11, 28], 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 [23], 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 [22, 5, 31], or an approach using symbolic execution to discover new tests satisfying certain constraints on the execution path [7, 6, 2]. A similar framework uses reinforcement learning, but constructs each test on-the-fly and performs update calls after every step of testing [12]. 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

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

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 [31, 25, 21, 33, 27]. Seed tests are usually manually generated, or tests selected from a previously generated suite for their high coverage or fault detection [30, 9]. It is generally the case that seed tests are more easily understood by users than newly generated tests. For example, seed tests often include only valid inputs and "reasonable" sequences of test actions, while generated tests, to improve coverage and fault detection, often include invalid inputs or bizarre method call sequences.

For example, consider the extremely popular and successful American Fuzzy Lop (AFL) tool for security fuzzing [31]. It usually begins fuzzing (trying to generate inputs that cause crashes indicating potential security vulnerabilities) from a corpus of "good" inputs to a program, e.g., actual audio or graphics files. 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 kind of trace back to the source of a generated test in some seed test (possibly through a long trail of also-generated tests) is essentially a *provenance*, which we argue is the most easily understood explanation of a learning result for humans, in those cases (such as testing) where the algorithm's purpose is to produce novel, interesting objects from existing objects.

This simple approach used in AFL 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 [32] 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 [19, 16, 17, 18] test generation language and tool for Python. In this approach, the seed tests are split into (usually 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 when to learn from a test, in such methods), it too is broken into sub-sequences and the sequences are added to the sub-sequence pool and used in generation of future tests.

In TSTL, a test is a sequence of components (test actions), and the provenance of a test generated using this algorithm involves numerous tests, and varying parts of those tests. We extended TSTL

```
int1 = 13                   # STEP 0   ;;; quick1.test:11
int0 = 7                    # STEP 1   ;;; quick1.test:14
int2 = 16                   # STEP 2   ;;; quick2.test:4
avl1 = avl.AVLTree()        # STEP 3   ;;; quick5.test:3
avl1.insert(int2)           # STEP 4   ;;; quick0.test:15
avl1.insert(int1)           # STEP 5   ;;; quick0.test:16
avl0 = avl.AVLTree()        # STEP 6   ;;; quick3.test:1
int1 = 10                   # STEP 7   ;;; quick3.test:2
avl0.insert(int0)           # STEP 8   ;;; quick3.test:3
avl0.insert(int1)           # STEP 9   ;;; quick3.test:4
avl0.delete(int0)           # STEP 10  ;;; quick3.test:5
avl1.insert(int2)           # STEP 11  ;;; quick5.test:10
int2 = 14                   # STEP 12  ;;; quick5.test:11
```

Figure 1: Example test generated with fine-grained provenance information

to allow every component of a test to be annotated with a string. Whenever a component from a sequence in the sequence pool is added to a test, it is labeled with the file name of the source test and the exact location in that test of the component. Figure 1 shows part of a high-quality test for an AVL data structure library. In the example, we first generated a set of "quick tests" [10, 9], small tests that together obtain maximum coverage. We then used sequences from these tests to guide testing, with the goal of producing a single test with maximal code coverage (the highest coverage from any one quick test is 173 branches and 131 statements). The complete generated test of which the first fragment is shown in Figure 1 covers 204 branches and 152 statements. Each step (component) in the test is labeled with its exact source in one of the 6 seed tests. Because all tests generated (including new quick tests enhancing coverage) are thus annotated, the source of a test component can always be traced back to an initial seed test. Here, the file names of the tests are not highly informative; however, using a recently-proposed technique for automatically giving generated tests high-quality names [4], the information could be even more useful. In practice, many seed tests would also be named for previously detected faults they are associated with, thus providing considerable information as to the context of test components.

In this example algorithm, provenance is certain and direct: each component of a test arises from one previously-generated or seed test. However, in some cases (e.g., learned models) multiple tests may influence the *probability* of a component appearing. For example, the probability of each possible component could be proportional to how many seed (and learned) tests that component appears in. In such cases, however, the same approach applies, except that instead of a single source, each component is labeled with a set of contributing tests, along with their degree of contribution (e.g., if a component appears 5 times in some seed tests, it will increase probability of generating that component more than a seed test in which it only appears once).

3 Presenting Collective Provenance Information

While provenance of components of a single test is the most interesting information for debugging and understanding a newly generated test, the most difficult and frequently performed part of an automated testing effort is the effort to understand the overall behavior of the test generation. For that task, information beyond the provenance of single tests is essential.

4 Test Manipulations and Pseudo-Provenance

The approaches presented above are applicable to a large set of test generation methods. However, tests generated are often manipulated after generation. In particular, they are very often *reduced* in size [32], producing a test with fewer components that preserves either fault detection or code coverage [10] properties of the original test. Reduction usually preserves provenance in a sense (components not removed maintain their provenance annotations), but a long sequence, as in our example algorithm, may no longer exist. Understanding the provenance after reduction is likely harder, because there are fewer long sequences from existing tests. Moreover, test *normalization* [13], which uses term rewriting to change (vs. simply remove) test components tends to destroy provenance information completely (any components rewritten during normalization lose their provenance, since the new component is produced by a brute force search, not from seed tests).

4.1 A Greedy Pseudo-Provenance Algorithm

5 Conclusions

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] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science*, pages 320–329, 2004.

[4] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 57–67, New York, NY, USA, 2017. ACM.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[11] Alex Groce, Alan Fern, Martin Erwig, Jervis Pinto, Tim Bauer, and Amin Alipour. Learning-based test programming for programmers. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 752–786, 2012.

[12] Alex Groce, Alan Fern, Jervis Pinto, Tim Bauer, Amin Alipour, Martin Erwig, and Camden Lopez. Lightweight automated testing with adaptation-based programming. In *IEEE International Symposium on Software Reliability Engineering*, pages 161–170, 2012.

[13] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 1–11, New York, NY, USA, 2017. ACM.

[14] 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.

[15] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 357–370, London, UK, UK, 2002. Springer-Verlag.

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

[17] 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.

[18] Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal, Josie Holmes, and Kevin Kellar. TSTL: the template scripting testing language. `https://github.com/agroce/tstl`, May 2015.

[19] 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.

[20] Mark Last, Abraham Kandel, and Horst Bunke. *Artificial intelligence methods in software testing*, volume 56. World Scientific, 2004.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. In *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'07, pages 136–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] 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.

[28] Richard Sutton and Andrew Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[29] 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.

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

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

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

[33] 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.