Evolutionary Fuzzing for Genetic Improvement: Toward Adaptive Software Defense

Jason Landsborough, Stephen Harding, Bryan Beabout
SSC Pacific, San Diego, California

ACM Reference format:
Jason Landsborough, Stephen Harding, Bryan Beabout. 2018. Evolutionary Fuzzing for Genetic Improvement: Toward Adaptive Software Defense. In *Proceedings of GI'18:IEEE/ACM 4th International Genetic Improvement Workshop , Gothenburg, Sweden, June 2, 2018 (GI'18),* 2 pages.
https://doi.org/10.1145/3194810.3194819

1 ABSTRACT

As fuzz testing strategies have become more and more sophisticated, we see a natural application of fuzz testing to Genetic Improvement techniques. In particular, the ability to generate high quality and high coverage tests with advanced fuzzers can greatly enhance the effectiveness of Genetic Improvement algorithms—especially when the algorithm is applied to bug fixing or other similar kinds of software improvement to improve qualities such as security.

2 INTRODUCTION

2.1 Fuzzers

Over the decades the security community has benefited greatly from fuzzing, a testing technique that probes software with invalid or random data [1] to find vulnerabilities. The techniques have grown significantly over the years, resulting in different classes of fuzzers. Blackbox fuzzers are the more traditional fuzzers, which repeatedly send random input and observe the program for crashes. Whitebox fuzzers have full knowledge of the program structure, and can guide input generation based on symbolic execution. Microsoft has been using their own proprietary whitebox fuzzer called Sage for years [2], and it found roughly one third of all bugs discovered during the development of Windows 7. Security researchers often do not have access to the source code for the software they are testing. There exists a middle ground in the most recent category of fuzzers, greybox fuzzing. Greybox fuzzers typically do not assume knowledge of the program through source code or sophisticated program analysis. Instead they utilize some form of instrumented feedback, such as code coverage. More recently, evolutionary greybox fuzzers [3] have been become popular. Evolutionary fuzzers utilize a genetic algorithm to mutate and optimize test inputs. Their fitness functions are often influenced by the amount of code covered (hence the greybox categoration) and crashes.

GI'18, June 2, 2018, Gothenburg, Sweden
2018. ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194819

2.2 Genetic Improvement

Similarly, over the past couple of decades, there has been research exploring software optimization and specialization through Genetic Improvement [4], an increasingly popular search-based software engineering approach using evolutionary algorithms. Genetic Improvement distinguishes itself from Genetic Programming as it starts with an existing program that will then be modified. It has been used for reducing energy consumption [5, 6], software specialization [7–9], performance [7, 10, 11], and automated bug fixing [12–14]. Genetic Improvement is most commonly used for modifying source code, but it has even been used on assembly language programs [15] and program binaries [9, 14]. In all cases, Genetic Improvement requires test cases to ensure correctness, which may limit its effectiveness when test suites are not sufficiently robust.

3 BUG HUNTING AND FIXING

Automated bug hunting using fuzzers and automated bug repair using Genetic Improvement seem a natural pairing. Haraldsson et al [16] implemented an automated bug fixing system inspired from Harman et al's dreaming device [17], using exceptions raised during user interaction during the day as test input for bug fixing using genetic improvement after work hours, finding 22 bugs over the course of 6 months. We believe that modern evolutionary fuzzers, possibly utilizing search-based software testing techniques [18], would compliment this approach to find even more bugs unlikely to be uncovered by a non-malicious user that can be used to automatically repair programs.

An exemplary example of a modern fuzzer which has received a lot of attention and use is the open-source fuzzer AFL, American Fuzzy Lop (also the name for a breed of rabbit), developed by Michal Zalewski [19]. At the time this paper was written, its bug-o-rama trophy case referenced 371 notable vulnerabilities. Because many bugs go unreported, such as those for internally-maintained software, the number of vulnerabilities is likely much higher. AFL supports blackbox fuzzing of binaries using QEMU, an open source machine emulator and virtualizer, or using a compiler flag instrumentation option for gcc or clang if source code is available. Forks of AFL exist to support other languages as well as kernel system calls [20] and virtual machines [21].

AFL is considered application-unaware in that it does not depend on application specific properties or data types. In contrast, a newer open source application-aware fuzzer, Vuzzer [22], released in 2017, uses static and dynamic analysis to learn properties of the application. For our purposes, we have considered the capabilities of AFL, which Vuzzer may also possess.

AFL is designed to be simple and it uses a genetic algorithm to guide fuzzing tests which explore new paths in the binary. Feedback to the genetic algorithm is measured in the form of coverage, consisting of branches hit, execution path, and crashes (in the form of segmentation faults). This results in the genetic algorithm favoring cases with new branches hit, or hit in a new order. Test cases that result in crashes are written out to disk so they can be analyzed later. Specifics on the internals of AFL can be found in AFL's technical whitepaper.

One of the powerful capabilities of evolutionary fuzzers, such as AFL, is evolving a large number of test cases that explore the input space of a program. Interestingly, evolutionary fuzzers have been able to learn additional protocol features or commands for protocols such as SMTP [3] and even learn the the JPEG filetype [23] with very minimal seed test input; just a file with the string "hello" in the later case.

We have begun initial experimentation with AFL and it shows promise in finding bugs and increasing test case coverage.

As an initial test we used a publicly-available vulnerable C program, based on a json parser consisting of numerous memory corruption bugs, called fuzzgoat [24]. The seed consisted of the invalid json string: {"":"}. After 43 hours, 67 unique crashes were found resulting in the same number of crash test cases. Often these consist of special characters, such as unicode and control characters, and very long strings. Additionally, it had 944 test cases queued for additional testing or mutation if it had continued to run. Using kcov [25], we measured the initial seed test case covering 29.4% of the executable code. With the 1,011 test cases, we measure 90.5% code coverage. With these new test cases, we can increase our assurance that transformations are safe, either online as part of the fitness function or offline as a sanity check, and can incorporate crashing test cases as part of the fitness test suite, which if repaired by no longer resulting in crashes would decrease the likelihood of an adversary exploiting those bugs.

Genetic Improvement, as an automated search technique relies heavily on the availability and quality of test suites to ensure mutated program validity. Evolutionary fuzzers use code coverage as a fitness metric, which results in more or better test cases, which we can leverage to increase assurance of correctness that modifications to a program do not have unintended side effects. Additionally, the primary purpose of fuzzers is to uncover new defects. These defects could then be automatically collected, then supplied as new test cases using Genetic Improvement to automatically repair the bugs.

4 FUTURE WORK

There are likely to be bugs which cannot be uncovered by a particular fuzzer. This could be due to exploitation of the input space rather than exploration, or simply not identifying a type of bug due to fuzzer implementation (fuzzers tend to be designed to discover crashes which can result in exploits). As computational resources become more cost effective, we can imagine a larger ensemble of application-aware and unaware fuzzers, search-based software testing based on program analysis, and user-driven bug reports getting us closer to achieving software which is able to adapt for defense.

There may also be opportunities to leverage recent progress in autonomic computing [26] to further enable self-* properties (self-configuration, self-healing, self-optimization, self-protection). Automated bug hunting and bug fixing could be a components within an autonomic system that can be controlled or customized based on system goals and priorities.

REFERENCES

[1] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[2] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[3] Jared DeMott, Richard Enbody, and William F Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing.

[4] Justyna Petke, Saemundur Haraldsson, Mark Harman, David White, John Woodward, et al. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[5] Bobby R Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1327–1334. ACM, 2015.

[6] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

[7] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[8] Kwaku Yeboah-Antwi and Benoit Baudry. Embedding adaptivity in software systems using the ecselr framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 839–844. ACM, 2015.

[9] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the kitchen sink from software. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 833–838. ACM, 2015.

[10] David R White, Andrea Arcuri, and John A Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[11] William B Langdon and Mark Harman. Grow and graft a better cuda pknotsrg for rna pseudoknot free energy calculation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 805–810. ACM, 2015.

[12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[13] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[14] Eric M Schulte, Westley Weimer, and Stephanie Forrest. Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 847–854. ACM, 2015.

[15] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316. ACM, 2010.

[16] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1513–1520. ACM, 2017.

[17] Mark Harman, Yue Jia, William B Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic improvement for adaptive software engineering (keynote). In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 1–4. ACM, 2014.

[18] Phil McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[19] Michal Zalewski. American fuzzy lop, 2015.

[20] Vegard Nossum and Quentin Casasnovas. Filesystem fuzzing with american fuzzy lop. In *Vault 2016*, Raleigh, N.C., April 21 2016.

[21] Jesse Hertz and helped by Tim Newsham. Project triforce: Run afl on everything!, 2016.

[22] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. 2017.

[23] Michal Zalewski. Pulling jpegs out of thin air, November 07 2014.

[24] David Moore. Kcov - code coverage, June 06 2017.

[25] Simon Kagstrom. fuzzgoat - a vulnerable c program for testing fuzzers, January 9 2018.

[26] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.