Editors: D. Balzarotti, davide.balzarotti@eurecom.fr | W. Enck, whenk@ncsu.edu | T. Holz, thorsten.holz@rub.de | A. Stavrou, astavrou@gmu.edu

The Rode0day to Less-Buggy Programs

Andrew Fasano | MIT Lincoln Laboratory and Northeastern University
Tim Leek | MIT Lincoln Laboratory
Brendan Dolan-Gavitt | New York University
Josh Bundt | Army Cyber Institute and Northeastern University

Despite their best efforts, computer programmers consistently fail to build the safe and reliable programs they imagine; rather, they typically build systems that work well until something unexpected happens. When presented with unexpected inputs, systems may reveal bugs in their code, which cause incorrect behavior or crashes. Every programmer knows that bugs are a persistent and costly issue in computer programs, but the path to getting rid of them is less clear.

Straightforward approaches such as manual code review are difficult to scale and often unsuccessful. Increasingly, developers are turning to automated bug-finding tools.

Static analyzers may automatically flag suspicious-looking code patterns or use more sophisticated analyses to discover potential vulnerabilities in large codebases. Another approach is to find bugs through dynamic testing (i.e., passing a large number of diverse inputs to a program and fixing any bugs that are discovered). This technique is overwhelmingly favored for both offensive and defensive security research, because this method is easy to perform with off-the-shelf tools, and it gives concrete results without false positives. Anyone can download an industrial-strength, coverage-guided fuzzer such as "American fuzzy lop" (AFL), and it will likely be able to test his or her programs. More importantly, this type of testing absolutely works; despite its easy setup, AFL (http://lcamtuf.coredump.cx/afl/) often succeeds at finding bugs.

But as scientists and practitioners, we would like to know not just that a tool like AFL works, but how well it works. We want to gauge bug-finding merit and improvement to guide the use and development of these tools. Is AFL better than KLEE, which uses symbolic execution and Satisfiability Modulo Theory solving to find bugs? Is this new version of AFL better than the last one? To answer such questions, we propose the following approach to testing bug-finding systems:

1. Create a corpus of buggy programs.
   - A corpus should contain multiple real-world programs.
   - Each of the programs should contain diverse and realistic bugs.
   - Each bug should manifest at a known location.
   - An input to trigger each bug should be generated.
2. Conduct an evaluation of bug finders.
   - At the start of an evaluation, all bug finders should be provided with the buggy programs from the same bug corpus. Other data in the corpus (e.g., triggering inputs and so on) should not be provided. Prior to the evaluation, this corpus must not be available in any form to developers of systems being evaluated.
   - Each bug finder should be evaluated based upon its ability to find inputs that trigger distinct bugs in the corpus.
3. After each evaluation, the full corpus of buggy programs should be released to the public with a list of bugs, where they manifest, and the triggering inputs for each.
4. Evaluations should be run repeatedly and frequently.

Over the past year, we have been running Rode0day [a portmanteau of a rodeo and a 0-day vulnerability, pronounced "ro-dee-oh-day" (http://rode0day.mit.edu)], a monthly bug-finding competition and our first attempt at this kind of precise bug-finder evaluation. We think this competition satisfies all of our criteria, with the possible exception of realistic bugs.

Bug Injection

To evaluate bug-finding systems, we first need buggy programs. In fact, we need tons of buggy programs for every evaluation. Manually creating

Digital Object Identifier 10.1109/MSEC.2019.2933682
Date of current version: 29 October 2019

 Copublished by the IEEE Computer and Reliability Societies

these programs clearly can't scale to the demand, but fortunately, recent research has shown that buggy programs can be created automatically. Previously, we created two systems to do just that: LAVA (which stands for "large-scale automated vulnerability addition") and Apocalypse,[3] which both automatically inject bugs into C programs.

LAVA uses a dynamic taint analysis to identify program variables controllable by input data. New code is injected into the program, which, depending on what those variables are set to, can cause a crash. The taint analysis reveals how values in the input file affect program variables; this information is used to generate input files that likely trigger each bug. LAVA combines this analysis with an empirical test to see if an input actually causes a program to crash. LAVA was presented at the IEEE Symposium on Security and Privacy in 2016, and the corpora from this article, LAVA-1 and LAVA-M, were released later the same year. At that time, the combined detection rate for a state-of-the-art fuzzer and symbolic execution bug finder was 2%. Since then, 47% of the bug-finding systems presented at the top four academic computer security conferences have self-evaluated against the LAVA-M corpus. However, the expiration date of LAVA-M seems to have come and gone, as evidenced by the Angora fuzzer[1] finding all of the labeled bugs in this corpus (and a few unlabeled ones, too).

Apocalypse employs symbolic execution and constraint solving as well as program synthesis to achieve a similar end. The bugs inserted by Apocalypse are fewer in number but far more complicated and stateful than are the LAVA ones. Although these bugs are harder to generate and fewer in quantity, they may better reflect the complexity of some bugs in real software.

Rode0day

With access to LAVA and Apocalypse, we decided to launch a recurring bug-finding competition for which we would release a corpus of buggy programs and challenge teams to find as many of the injected bugs as they could. Each competition would run for a month, during which time teams would try their best to find bugs. At the end of each competition, we would declare a winner and release inputs to trigger each of the injected bugs. Our vision of how this competition improves bug finding is shown in Figure 1.

At the start of each Rode0day, teams are given a collection of Linux programs, sample inputs, and instructions about how to run each program on the sample inputs. Teams use their bug-finding skills to generate inputs that cause target programs to crash. When a team believes they have generated such an input, they submit it to the Rode0day API and are scored immediately. For each new bug a team finds, they are awarded 10 points. If a team is the first to find a given bug, they are awarded a single bonus point. This encourages teams to submit their generated inputs as soon as they find them. As teams submit new inputs to the API, their scores are updated in real time on a scoreboard on the Rode0day website. Although we conceal from competitors the total number of bugs that were injected into a given challenge, the scoreboard does show the number of unique bugs found by all the teams, thus providing a lower bound.

In May 2018, we ran a closed beta Rode0day competition. Using LAVA, we injected 52 bugs into two simple C programs we had written: a trivial file parser and a simple key/value store. On average, our nine competitors found slightly more than half of the injected bugs. The first-place team, itszn, found every injected bug in roughly 5 h, an indication that future competitions should feature more substantial programs.

We launched our first official Rode0day competition in July 2018 and have been running new competitions (nearly) every month since. At the time of this writing, we have run 10 competitions with 44 buggy programs containing a total of 2,103 bugs. These programs are generated from the real software displayed in Table 1. A total of 28 teams have scored points in these competitions.

Results

Administering Rode0day provides us with a wealth of empirical data about what works and what doesn't for finding bugs. It also allows us to conduct controlled experiments by introducing different types of bugs, varying the size of programs, and using different bug-injection strategies. Although we haven't yet conducted a full analysis of the data we've collected in the past year, in this article, we provide a sampling of some of our more interesting results.

One obvious question is: Which team is the best? Answering this question is not completely straightforward, because not every team has participated in every competition. To assess the relative performance of different teams, we used a version of the Elo rating system, which was developed to rank chess players. Specifically, we use the multiplayer variant of Elo described by Tom Kerrigan at http://www.tckerrigan.com/Misc/Multiplayer_Elo. In the Elo system, every player starts out with the same skill rating. When two players play a match against each other, the difference in their scores is used to calculate the expected outcome. The difference between the expected and actual outcome is then used to update their ratings. The rankings for our first year are shown in Table 2. The top-ranked team, afl-lazy, played in the last five competitions and won four of them. Despite being the best team to participate thus far, afl-lazy has found fewer than half the injected bugs across all

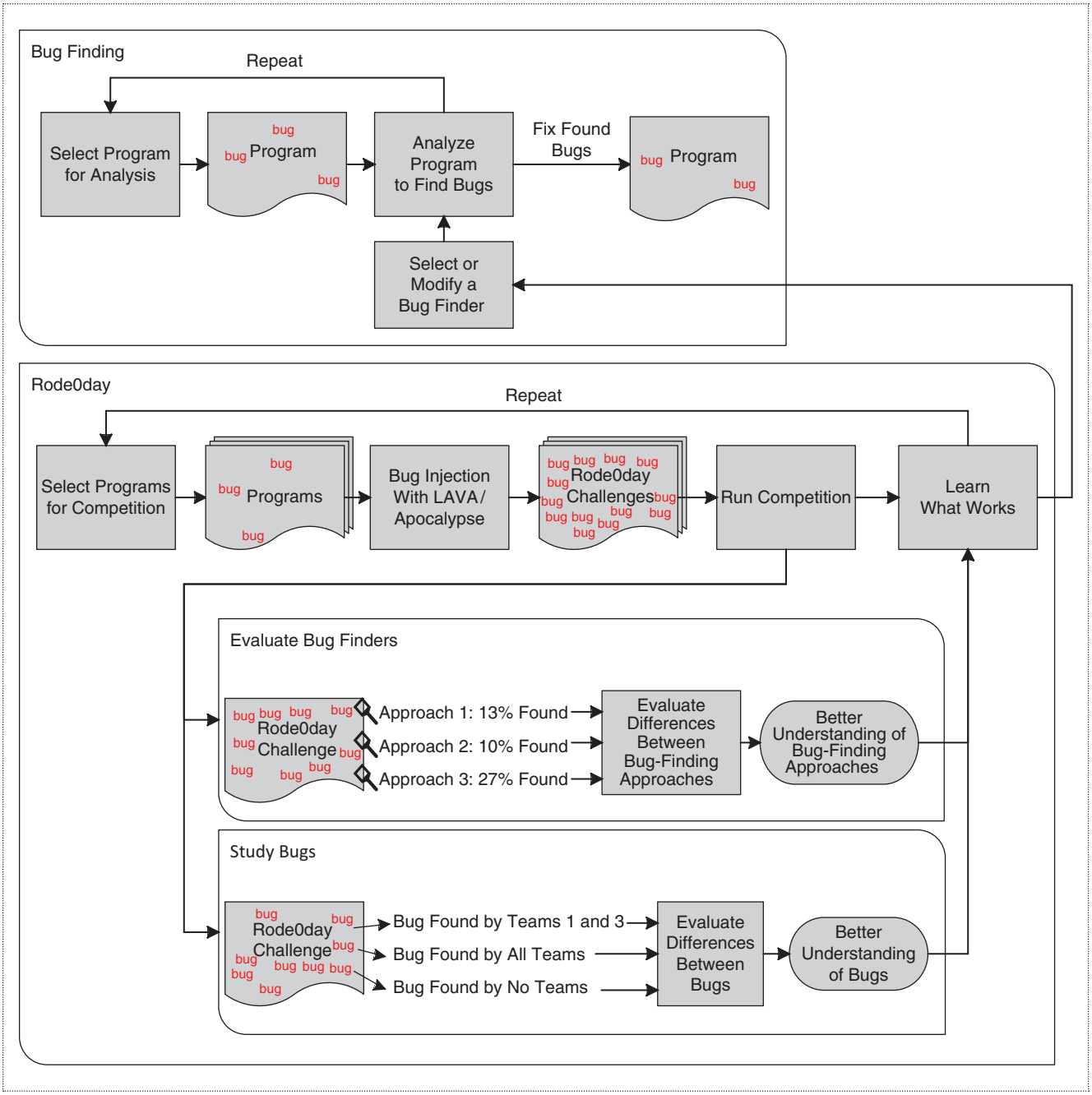

Figure 1. An example of how Rode0day improves bug finding.

the Rode0day competitions they participated in. Finding these injected bugs is clearly not a solved problem.

In addition to comparing team performances, we're also interested in studying what makes some bugs easy to find and others difficult. Figure 2 shows which bugs were found in a version of a file used in the November 2018 Rode0day. For this challenge, the 59 LAVA bugs injected were of two types: simple bugs, in which a memory safety violation occurs if a value from the input matches a constant; and complex bugs, in which a memory safety violation occurs if three values from the input satisfy a particular equation. Fifty percent of the 36 injected simple bugs and 70% of the 23 injected complex bugs were found by the six competing teams. Each column in Figure 2 corresponds to a discovered bug. The color of each square indicates the time it took to find a bug: green squares indicate bugs found within 24 h, light blue squares indicate bugs that took just over a day, and darker blue squares took longer (up to 25 days). Of these discovered bugs, some are clearly

Table 1. The target programs used in Rode0day competitions.

Name	Description	C-SLOC*	Bugs injected	Bugs discovered
File	File parser	15,671	657	431
LibYAML	YAML parser	10,324	231	216
Libjpeg	JPEG parser	20,316	268	268
Bzip2	File compressor	5,820	28	28
Duktape	JavaScript engine	83,793	18	15
Grep	Regular expression matcher	70,030	80	78
Jq	JSON parser	16,897	332	210
Pcre2grep	Regular expression matcher	83,113	184	180
Sqlite	Database	12,902	56	24
Tinyexpr	Math engine	1,241	44	37

**Lines of C code as measured by David A. Wheeler's SLOCCount.*
SLOC: source lines of code.

Table 2. The overall rankings for top Rode0day competitors after 10 competitions.

Place	Elo score	Team name
1	1,087	afl-lazy
2	1,069	itszn
3	1,027	H3ku
4	1,017	REDQUEEN
5	1,062	NU-AFL-QSYM

easier to find than others, as evidenced by multiple teams finding the same bugs within a day.

Of the 34 discovered bugs shown, the rightmost column represents a bug found by a single team after 25 days of searching. To understand some of the challenges involved in finding this bug, let's walk through what needs to occur for it to be triggered. The bug requires three separate program variables (i.e., *A*, *B*, and *C*) to be set to values copied from separate portions of the input file. If the values of *A*, *B*, and *C* satisfy the equation $((A * B) - C) == 0xf14ec3$, an index into an array is offset by the value from *B*, so it is possible (though difficult) to generate an input that will lead to an out-of-bounds array access. The input file also determines the path taken by the program, and this path must visit the locations in the code where *A*, *B*, and *C* are assigned values from the input. However, the sample input provided to competitors did not execute this code for *A* and *C*, nor did it reach the potentially invalid array access. Competitors had to find an input that could reach these parts of the program.

But these challenges are not unique to this bug; other bugs required solving similar equations and executing new parts of the program. So why was this bug so much harder to find than others? And why could NU_AFL_QSYM find it and other teams could not? As we continue to run our Rode0day competitions, we hope to collect enough data to answer questions like these.

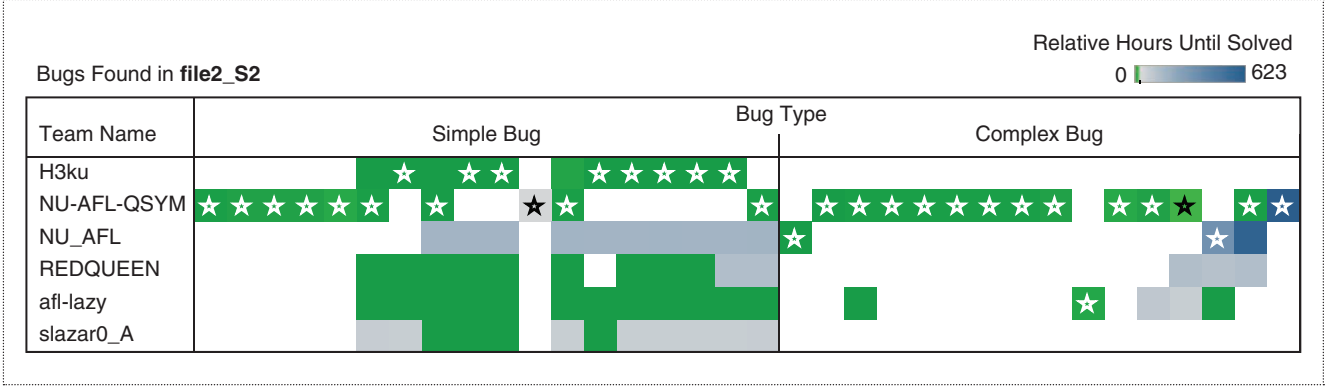

Figure 2. A visualization of when teams found bugs in *file2_S2* during the November 2018 Rode0day. Green indicates the bugs found within 24 h of a team's first score. The stars denote the first team to find a bug.

By finding a way to quantify how difficult a bug is to find with a given approach to bug finding, we hope to gain an understanding of how different bug-finding strategies can be used in combination or how new bug-finding systems can be developed to find and fix more bugs in real programs.

The first year of Rode0day has been a success, but we see several areas where it can be improved. Most pressingly, we have yet to ensure that the bugs used in the competition are representative of the naturally occurring bugs we'd like our bug finders to uncover. We plan to do this by 1) looking at many real-world bugs and trying to get LAVA to match their properties in ways that matter and 2) validating that the systems that perform well in our competition are also successful in the real world. We would also like to open up Rode0day to other bug-injection systems aside from LAVA and Apocalypse, such as hand-created bug corpora. This would allow others to contribute bugs that they think are more realistic or that cover new bug classes (e.g., use after free) that we haven't had the time to implement.

Another limitation is that most static analysis tools cannot currently participate because they do not provide a triggering input that demonstrates they have found a given bug. For each of the injected bugs, however, we know exactly where their "root cause" is, and because our bugs are injected into source code, we can, in principle, evaluate static analyzers as well. This task is slightly more complicated than evaluating fuzzers, both because of the possibility of false positives and because different tools may reasonably disagree about the location of a bug in the program. For instance, if a pointer is corrupted on one line of a program but does not cause a crash until it is dereferenced on another line, which line contains the bug? We think these challenges can be overcome with careful scoring design and some empirical examination of how current static analyzers report bugs.

Finally, for the time being, we have intentionally made no attempt to place computational restrictions on competitors. This means that a well-funded team could potentially perform better by dedicating thousands of cores to bug finding. It also makes it more difficult to compare the performance of different tools. To make more direct, apples-to-apples comparisons between different tools, we plan to create a yearly, computation-limited competition. For this event, competitors would submit a Docker container or virtual machine containing their bug-finding system; we would then allocate some fixed amount of resources on a standard cloud platform like Amazon EC2 to run the bug-finding systems and declare a winner.

As we continue running Rode0day, we hope to keep improving our competitions to provide the best possible evaluation of bug-finding systems. Along the way, we'll keep collecting, publishing, and analyzing our data in the hope of giving our community a better understanding of bugs and bug finding. ■

Acknowledgments

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under U.S. Air Force contract FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

References

1. P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Security and Privacy*, 2018, pp. 711–725.
2. B. Dolan-Gavitt et al., "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Security and Privacy*, 2016, pp. 110–121.
3. S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proc. ACM Joint Meeting European Software Engineering Conf. Symp. Foundations of Software Engineering*, 2018, pp. 224–234.

Andrew Fasano is a member of the research staff at the MIT Lincoln Laboratory, Lexington, Massachusetts, and a Ph.D. student at Northeastern University, Boston, Massachusetts. Fasano received a bachelor's degree from Rensselaer Polytechnic Institute, Troy, New York. Contact him at fasano@mit.edu.

Tim Leek is a member of the research staff at the MIT Lincoln Laboratory, Lexington, Massachusetts. Leek received a master's degree from the University of California San Diego. He is a member of USENIX and the Association for Computing Machinery. Contact him at tleek@ll.mit.edu.

Brendan Dolan-Gavitt is an assistant professor at New York University. Dolan-Gavitt received a Ph.D. from the Georgia Institute of Technology, Atlanta. He is a Member of the IEEE. Contact him at brendandg@nyu.edu.

Josh Bundt is a Ph.D. student at Northeastern University, Boston, Massachusetts. Bundt received a master's degree from the Naval Postgraduate School, Monterey, California. He is a member of USENIX. Contact him at jmb@ccs.neu.edu.