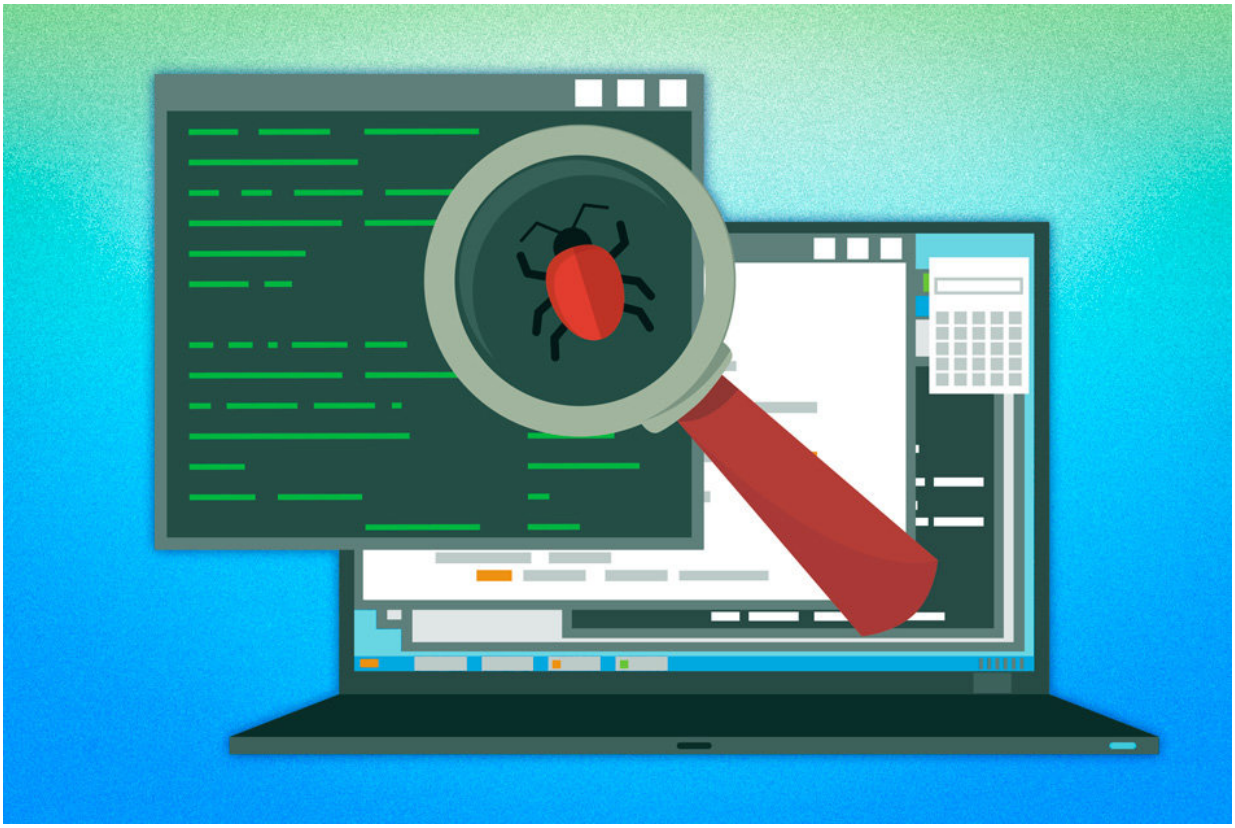
Bug-repair system learns from example

September 29 2017, by Larry Hardesty



A new machine-learning system analyzes successful repairs to buggy software and learns how to repair new bugs. Credit: Massachusetts Institute of Technology

Anyone who's downloaded an update to a computer program or phone app knows that most commercial software has bugs and security holes that require regular "patching."

Often, those bugs are simple oversights. For example, the program tries to read data that have already been deleted. The patches, too, are often simple—such as a single line of code that verifies that a data object still exists.

That simplicity has encouraged computer scientists to explore the possibility of automatic patch generation. Several research groups, including that of Martin Rinard, an MIT professor of electrical engineering and computer science, have developed templates that indicate the general forms that patches tend to take. Algorithms can then use the templates to generate and evaluate a host of candidate patches.

Recently, at the Association for Computing Machinery's Symposium on the Foundations of Software Engineering, Rinard, his student Fan Long, and Peter Amidon of the University of California at San Diego presented a new system that learns its own templates by analyzing successful patches to real software.

Where a hand-coded patch-generation system might feature five or 10 templates, the new system created 85, which makes it more diverse but also more precise. Its templates are more narrowly tailored to specific types of real-world patches, so it doesn't generate as many useless candidates. In tests, the new system, dubbed Genesis, repaired nearly twice as many bugs as the best-performing hand-coded template system.

Thinning the herd

"You are navigating a tradeoff," says Long, an MIT graduate student in electrical engineering and computer science and first author on the paper. "On one hand, you want to generate enough candidates that the set you're looking through actually contains useful patches. On the other hand, you don't want the set to include so many candidates that you can't search through it."

Every item in the data set on which Genesis was trained includes two blocks of code: the original, buggy code and the patch that repaired it. Genesis begins by constructing pairs of training examples, such that every item in the data set is paired off with every other item.

Genesis then analyzes each pair and creates a generic representation—a draft template—that will enable it to synthesize both patches from both originals. It may synthesize other, useless candidates, too. But the representation has to be general enough that among the candidates are the successful patches.

Next, Genesis tests each of its draft templates on all the examples in the training set. Each of the templates is based on only two examples, but it might work for several others. Each template is scored on two criteria: the number of errors that it can correct and the number of useless candidates it generates. For instance, a template that generates 10 candidates, four of which patch errors in the training data, might score higher than one that generates 1,000 candidates and five correct patches.

On the basis of those scores, Genesis selects the 500 most promising templates. For each of them, it augments the initial two-example training set with each of the other examples in turn, creating a huge set of three-example training sets. For each of those, it then varies the draft template, to produce a still more general template. Then it performs the same evaluation procedure, extracting the 500 most promising templates.

Covering the bases

After four rounds of this process, each of the 500 top-ranking templates has been trained on five examples. The final winnowing uses slightly different evaluation criteria, ensuring that every error in the training set that can be corrected will be. That is, there may be a template among the final 500 that patches only one bug, earning a comparatively low score in

the preceding round of evaluation. But if it's the only template that patches that bug, it will make the final cut.

In the researchers' experiments, the final winnowing reduced the number of templates from 500 to 85. Genesis works with programs written in the Java programming language, and the MIT researchers compared its performance with that of the best-performing hand-coded Java patch generator. Genesis correctly patched defects in 21 of 49 test cases drawn from 41 open-source programming projects, while the previous system patched 11.

It's possible that more training data and more computational power—to evaluate more candidate templates—could yield still better results. But a system that allows programmers to spend only half as much time trying to repair bugs in their code would be useful nonetheless.

More information: Automatic Inference of Code Transforms for Patch Generation. people.csail.mit.edu/rinard/pa … er/fse17.genesis.pdf

This story is republished courtesy of MIT News (web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.

Provided by Massachusetts Institute of Technology