Dude, where's my code?

October 16 2013, by Larry Hardesty

Compilers are computer programs that translate high-level instructions written in human-readable languages like Java or C into low-level instructions that machines can execute. Most compilers also streamline the code they produce, modifying algorithms specified by programmers so that they'll run more efficiently.

Sometimes that means simply discarding lines of code that appear to serve no purpose. But as it turns out, compilers can be overaggressive, dispensing not only with functional code but also with code that actually performs vital security checks.

At the ACM Symposium on Operating Systems Principles in November, MIT researchers will present a new system, dubbed Stack, that automatically combs through programmers' code, identifying just those lines that compilers might discard but which could, in fact, be functional. Although the paper hasn't appeared yet, commercial software engineers have already downloaded Stack and begun using it, with encouraging results.

As strange as it may seem to nonprogrammers—or people whose only experience with coding is on small, tightly managed projects—large commercial programs are frequently full of instructions that will never be executed, known as "dead code." When hundreds of developers are working on an application with millions of lines of code that have been continually revised for decades, one of them may well end up inserting a seemingly innocuous condition that ensures that a function thousands of lines away, written by someone else, never gets executed. Dead code is ubiquitous, and compilers should remove it.

Problems arise when compilers also remove code that leads to "undefined behavior." "For some things this is obvious," says Frans Kaashoek, the Charles A. Piper Professor in the Department of Electrical Engineering and Computer Science (EECS). "If you're a programmer, you should not write a statement where you take some number and divide it by zero. You never expect that to work. So the compiler will just remove that. It's pointless to execute it anyway, because there's not going to be any sensible result."

Defining moments

Over time, however, "compiler writers got a little more aggressive," Kaashoek says. "It turns out that the C programming language has a lot of subtle corners to the language specification, and there are things that are undefined behavior that most programmers don't realize are undefined behavior."

A classic example, explains Xi Wang, a graduate student in EECS and first author on the new paper, is the assumption that if a program attempts to store too large a number at a memory location reserved for an integer, the computer will lop off the bits that don't fit. "In machines, integers have a limit," Wang says. "Whenever you exceed that limit, the input value basically wraps around to a smaller value."

Seasoned C programmers will actually exploit this behavior to verify that program inputs don't exceed some threshold. Rather than writing a line of code that, say, compares the sum of two numbers to the known threshold for an integer ("if a + b a")—whether, that is, the summation causes the integer to wrap around to a smaller value.

According to Wang, programmers give a range of explanations for this practice. Some say that the intent of the comparison—an overflow check—is clearer if they use integer wraparound; others say that the wraparound comparison executes more efficiently than the more conventional comparison; and some maintain that it avoids cluttering up their code with unneeded terminology (like "int_max"). But whatever the reason, while the wraparound check works fine with unsigned integers—integers that are always positive—it is, according to the C language specification, undefined for signed integers—integers that can be either positive or negative.

As a consequence, some C compilers will simply discard the wraparound comparison. And sometimes, that can mean dispensing with a security check that guarantees the program's proper execution.

The fine print

Complicating things further is the fact that different compilers will dispense with different undefined behaviors: Some might permit wraparound checks but prohibit other programming shortcuts; some might impose exactly the opposite restrictions.

So Wang combed through the C language specifications and identified every undefined behavior that he and his coauthors—Kaashoek and his fellow EECS professors Nickolai Zeldovich and Armando Solar-Lezama—imagined that a programmer might ever inadvertently invoke. Stack, in effect, compiles a program twice: once just looking to excise dead code, and a second time to excise dead code and undefined behavior. Then it identifies all the code that was cut the second time but not the first and warns the programmer that it could pose problems.

The MIT researchers tested their system on several open-source programs. In one case, the developers of a program that performs database searches refused to believe that their code had bugs, even after they'd examined the instructions flagged by Stack. "Xi sent them a one-line SQL statement that basically crashed their [application], by exploiting their 'correct' code," Kaashoek says.

Mattias Engdegård, an engineer at Intel, is one of the developers who found Stack online and has already applied it to his company's code. "Stack is very carefully designed to have a very low false-positive ratio," Engdegård says. Nonetheless, "it found some errors that no other static-analysis tool had found before," he says, resulting in "one or two dozens of instances of code changes."

"This could be some kind of harbinger of things to come," Engdegård adds. "I think static analyzers are going to focus on these sort of things in the future."

The paper is titled "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior."

More information: pdos.csail.mit.edu/~xi/papers/stack-sosp13.pdf

Provided by Massachusetts Institute of Technology

Citation: Dude, where's my code? (2013, October 16) retrieved 26 April 2024 from https://phys.org/news/2013-10-dude-code.html

This document is subject to copyright. Apart from any fair dealing for the purpose of private study or research, no part may be reproduced without the written permission. The content is provided for information purposes only.