

Technical Perspective

Patching Program Errors

By Martin C. Rinard

C PROGRAMMERS ARE all too familiar with out-of-bounds memory errors. A typical scenario involves a pointer calculation that produces an address outside the target block of memory that the developer intends the program to write. The resulting out-of-bounds writes corrupt the data structures of otherwise unrelated parts of the program, causing the program to fail or mysteriously generate unexpected outputs. Analogous errors in Java and other type-safe languages cause out-of-bounds exceptions, which typically terminate the execution of the program.

The paper here presents an intriguing technique for automatically isolating and correcting these errors. The basic idea is simple. Allocate blocks randomly within a memory much larger than required to execute the program. Set the remaining unused memory to contain specific “canary” values. Run multiple versions of the program to failure, then observe overwritten canary values to compute 1) the target block the out-of-bounds writes were intended to write, and 2) how much additional memory would have been required to bring the writes back within bounds. The fix is to go back to the allocation site in the program that allocated the target block, then apply a patch that suitably increases the size of blocks allocated at that site. The paper also presents a conceptually similar technique for dealing with accesses to prematurely deallocated blocks of memory. Note that the combination of random allocation and multiple heaps is the key insight that makes it possible to isolate the target block. Randomizing the block placement provides the block location diversity required to ensure that, with high probability, only the target block will have the same offset from the overwritten canary values in all of the heaps.

One interesting aspect of this approach is that it generates patches almost immediately and without the need for human interaction. It is therefore suitable for uses (such as eliminating security vulnerabilities) that place a premium on obtaining fast responses to newly exposed errors. It can also eliminate the need to interact with a

(potentially distracted, indifferent, recalcitrant, or defunct) software development organization to obtain relief from an error, even when the software is distributed only in binary form.

An even more interesting aspect of this approach is that it may very well produce patches that do not completely fix the problem—the fact that the addition of a given amount of memory would have eliminated the out-of-bounds accesses in one observed execution provides no guarantee that it will eliminate such accesses in other executions. Benefits of this approach include simplicity and feasibility of implementation and deployment.

Over the last several years I have had many conversations about this and other (more aggressive or even unsound) techniques for automatically correcting or tolerating errors. Many developers and researchers find something deeply unsettling about a program that continues to execute in the face of errors. In fact, the most common response is that programs should stop when they encounter an error and not continue until a developer fixes the error.


I believe I get this response for two reasons. First, most developers feel responsible for the behavior of the programs they produce. Automatic intervention can often invalidate, undermine, or simply bypass the reasoning the developer originally used when developing the program. It is not clear what is going to replace that reasoning or who is then responsible for the actions of the modified program. Second, the needs of the developer are usually best served with a fail-stop approach. Stopping the execution as close as possible to the error makes it easier for developers to isolate and fix errors while the program is under development. Many developers instinctively reject any approach that involves continued execution after an error, presumably because the continued execution can complicate debugging by obscuring the location of the error.

But the needs of users are very different from the needs of developers. Stopping the program at the first sign of an

error can unacceptably deny access to important functionality. The undesirability of this denial of service can be seen (in embryonic form) in the common practice of removing assertions before releasing a system for production use.

Even if you believe that programs that continue to execute through errors are more likely to produce unacceptable results than programs that execute without a detected error, there are clearly many scenarios in which continued execution is superior. Consider, for example, a program that produces a result (such as a drawing or digitally edited picture) whose acceptability is obvious upon inspection. Continued execution in the face of errors may very well enable the program to produce an acceptable result that satisfies the user’s needs.

A fail-stop approach can also be dangerous when applied to programs that control unstable physical phenomena. In this case, continued execution through errors can offer the only real hope of obtaining an acceptable outcome. To cite a real-world example, consider the infamous Ariane 5 disaster. This disaster was directly caused by the inappropriate use of a fail-stop approach in a safety-critical embedded software system.

So what does the future hold for automatic error recovery and repair? At this point we have a variety of viable strategies for dealing with errors (such as out-of-bounds memory accesses or null-pointer dereferences) that no program should ever commit. Future advances will focus on correcting application-specific errors. The key is to obtain specifications that provide a foundation for the recognition and elimination of unacceptable behavior. One particularly fruitful area is sure to be unsound techniques that (in return for simplicity and feasibility of implementation and deployment) ignore traditional requirements that program transformations should never perturb error-free executions. The success of such techniques could then pave the way for a more mature software engineering perspective that views correctness as simply one of a set of engineering trade-offs to be appropriately managed during the lifetime of the system. 

Martin C. Rinard (rinard@cag.csail.mit.edu) is a professor in the Department of Electrical Engineering and Computer Science at MIT, Cambridge, MA.