Automated Techniques for Surviving (Otherwise) Fatal Software Errors

Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
`rinard@csail.mit.edu`

1 Fatal Software Errors

Many errors in software systems do not manifest themselves until the system has been deployed into production use. Fatal errors can have especially severe consequences in such situations as they may completely deny the user any of the service that the program is designed to supply. The standard approach to dealing with errors is to notify the organization that produced the system of the problm, have a developer investigate the problem to discover and correct the error, then issue a patch or new release with the error corrected. A host of issues make this approach suboptimal:

- **Error Notification:** In many cases the software may be executing autonomously with no connection to the organization that produced it; in other cases the users of the software may be reluctant to notify the organization that produced the software of the error (for example, because notifying the organization of the error may reveal information that the users prefer to keep private). In these cases, and others, the organization that produced the software may never even become aware of the error.
- **Error Correction Delays:** Any process that involves developers maintaining software incurs inevitable delays as the developers investigate the error, correct the code that contains the error, and validate the correction. In the meantime the users of the system must wait for the maintenance process to complete. These delays can be especially intolerable, for example, in real-time systems that control unstable physical phenomena and in commercial applications that cannot tolerate substantial downtime.
- **Distribution Difficulties:** Once the error is corrected, the corrected version must be distributed to its users. This distribution can be especially problematic in embedded systems with no connectivity to the organization that corrected the error or if the deployment of the new version involves substantial system administration activities.
- **Error Reproduction:** The error may manifest itself only in certain operating contexts, making it difficult for the developer investigating the error to reproduce the error.

– **New Errors or Anomalies:** Attempts to correct the error may introduce new errors or change the behavior of the software in ways that disable other parts of the system or other uses of the system.

Together, all of these issues provide a strong incentive to develop and deploy techniques that enable systems to continue to execute through otherwise fatal errors. In many cases this incentive is strong enough to make users willing to take the risk of of impaired functionality or even unacceptable execution in return for the certainty of continued execution that may satisfy their needs.

2 Surviving (Otherwise) Fatal Software Errors

We have developed several techniques that, together, eliminate the possibility of certain kinds of fatal errors:

– **Forced Loop Termination:** Infinite loops are one source of fatal errors — they can prevent flow of control from proceeding past the loop to execute other crucial parts of the program. Our infinite loop elimination technique simply learns reasonable bounds for the number of iterations that each loop may execute by observing successful executions, then exits each loop if it attemps to substantially exceed its reasonable iteration bound. The potential drawback of this technique is that it may exit the loop prematurely; the advantage is that it completely eliminates the possibility of an infinite loop.
– **Forced Recursion Termination:** Infinite recursions are another potential source of fatal errors because they can exhaust the stack space. Our infinite recursion elimination technique simply bounds the size of the stack, then immediately returns back out of any procedure call that attempts to exceed this bound. The potential drawback is that it may terminate recursions prematurely; the advantage is that it can eliminate otherwise fatal infinite recursions.
– **Deadlock Elimination:** Our deadlock elimination technique simply goes around releasing locks until any deadlock is eliminated. The potential drawback is the introduction of race conditions because of unsynchronized accesses to shared data; the advantage is the elimination of deadlock.
– **Memory Leak Elimination:** Memory leaks can cause a program to fail because it exhausts its address space. Our memory leak elimination algorithm simply allocates a fixed size buffer for allocation sites that may leak allocated memory. It then allocates data cyclically out of that buffer. The potential drawback is that the technique may overlay live data; the advantage is the elimination of any memory leak at that site.
– **Resource Leak Elimination:** Memory is only one of the many resources that a system may need to successfully execute. Systems may also require file handles, mutual exclusion locks, condition variables, and other resources. Our resource leak elimination algorithms apply a generalization of our memory leak elimination idea — they allocate a conceptually unbounded number of resources out of fixed size pools by applying some policy for reallocating

resources out of the pools when the system exhausts its resources. Potential policies include least-recently-used reallocation and policies that attempt to estimate importance and reallocate resources in reverse importance order.
– **Invalid Addressing Elimination:** Out of bounds writes can cause fatal address space corruption; out of bounds reads can cause the program to attempt to access an invalid part of the address space. Our invalid addressing elimination technique (also known as failure-oblivious computing) performs bounds checks to discard out of bounds writes and manufacture arbitrary values for out of bounds reads. The potential disadvantage is that the program may be unprepared to operate with the manufactured read values; the advantage is the elimination of out of bounds accesses as the immediate cause of fatal errors.

Because all of these techniques change the execution of the program in potentially unpredictable ways, it seems that an empirical investigation is the only productive way to explore the potential effects they may have on the execution. We have performed such an investigation for several of these techniques. Our results indicate that software systems usually tolerate the perturbation that these techniques can introduce. The overall result is usually acceptable execution with, in some cases, a relatively graceful degradation in service.

References

1. H. Nguyen and M. Rinard. Using cyclic memory allocation to eliminate memory leaks. Technical Report MIT-LCS-TR-1008, MIT Laboratory for Computer Science, October 2005.
2. M. Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session*, Anaheim, CA, October 2003.
3. M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 2004 Annual Computer Security Applications Conference*, Tucson, AZ, December 2004.
4. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.
5. M. Rinard, C. Cadar, and H. Nguyen. Exploring the acceptability envelope. In *2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '05 Companion) Onwards! Session*, San Diego, CA, October 2005.