acmqueue

Software Needs Seatbelts and Airbags

Finding and fixing bugs in deployed software is difficult and time-consuming. Here are some alternatives.

Emery D. Berger, University of Massachusetts, Amherst

Like death and taxes, buggy code is an unfortunate fact of life. Nearly every program ships with known bugs, and probably all of them end up with bugs that are discovered only post-deployment. There are many reasons for this sad state of affairs.

One problem is that many applications are written in memory-unsafe languages. Variants of C, including C++ and Objective-C, are especially vulnerable to memory errors such as buffer overflows and dangling pointers (use-after-free bugs). Two of these are in the SANS Top 25 list: buffer copy without checking size of input (http://cwe.mitre.org/top25/index.html#CWE-120) and incorrect calculation of buffer size (http://cwe.mitre.org/top25/index.html#CWE-131); see also heap-based buffer overflow (http://cwe.mitre.org/data/definitions/122.html) and use after free (http://cwe.mitre.org/data/definitions/416.html).

These bugs, which can lead to crashes, erroneous execution, and security vulnerabilities, are notoriously challenging to repair.

SAFE LANGUAGES: NO PANACEA

Writing new applications in memory-safe languages such as Java instead of C/C++ would help mitigate these problems. For example, because Java uses garbage collection, Java programs are not susceptible to use-after-free bugs; similarly, because Java always performs bounds-checking, Java applications cannot suffer memory corruption caused by buffer overflows.

That said, safe languages are no cure-all. Java programs still suffer from buffer overflows and null pointer dereferences, although they throw an exception as soon as they happen, unlike their C-based counterparts. The common recourse to these exceptions is to abort execution and print a stack trace (even to a Web page!). Java is also just as vulnerable as any other language to concurrency errors such as race conditions and deadlocks.

There are both practical and technical reasons not to use safe languages. First, it is generally not feasible to rewrite existing code because of the cost and time involved, not to mention the risk of introducing new bugs. Second, languages that rely on garbage collection are not a good fit for programs that need high performance or that make extensive use of available physical memory, since garbage collection always requires extra memory.[6] These include operating-system-level services, database managers, search engines, and physics-based games.

ARE TOOLS THE ANSWER?

While tools can help, they cannot catch all bugs. Static analyzers have made enormous strides in recent years, but many bugs remain out of reach. Rather than swamp developers with false positive reports, most modern static analyzers report far fewer bugs than they could. In other words, they

trade false negatives (failing to report real bugs) for lower false positive rates. That makes these tools more usable, but it also means they will fail to report real bugs. Dawson Engler and his colleagues made exactly this choice for Coverity's "unsound" static analyzer.[4]

The state of the art in testing tools has also advanced dramatically in the past decade. Randomized fuzz testing can be combined with static analysis to explore paths that lead to failure. These tools are now in the mainstream: for example, Microsoft's Driver Verifier can test device-driver code for a variety of problems and now includes randomized concurrency stress testing.

As Dijkstra famously remarked, however, "Program testing can be used to show the presence of bugs, but never to show their absence!" At some point, testing will fail to turn up new bugs, which will unfortunately be discovered only after the software has shipped.

FIXING BUGS: RISKY (AND SLOW) BUSINESS

Finding the bugs is only the first step. Once a bug is found—whether by inspection, testing, or analysis—fixing it remains a challenge. Any bug fix must be undertaken with extreme care, since any new code runs the risk of introducing yet more bugs. Developers must construct and carefully test a patch to ensure that it fixes the bug without introducing any new ones. This can be costly and time consuming. For example, the average time between the discovery of a *remotely exploitable* memory error and the release of a patch for enterprise applications is 28 days, according to Symantec.[12]

At some point, fixing certain bugs simply stops making economic sense. Tracking their source is often difficult and time consuming, even when the full memory state and all inputs to the program are available. Obviously, showstopper bugs must be fixed. For other bugs, the benefits of fixing them may be outweighed by the risks of creating new bugs and the costs in programmer time and delayed deployment.

DEBUGGING AT A DISTANCE

Once the faulty software has been deployed, the problem of chasing down and repairing bugs becomes exponentially more difficult. Users rarely provide detailed bug reports that allow developers to reproduce the problem.

For deployed software on desktops or mobile devices, getting enough information to find a bug can be difficult. Sending an entire core file is generally impractical, especially over a mobile connection. Typically, the best one can hope for is some logging messages and a minidump consisting of a stack trace and information about thread contexts.

Even this limited information can provide valuable clues. If a particular function appears on many stack traces observed during crashes, then that function is a likely culprit. Microsoft Windows includes an application debugger (formerly Watson, now Windows Error Reporting) that is used to perform this sort of triage not only for Microsoft but also for third-party applications via Microsoft's Winqual program. Google also has made available a cross-platform tool called Breakpad that can be used to provide similar services for any application.

For many bugs, however, the kind of information that these tools provide is of limited value. For example, memory-corruption errors often do not trigger failures until millions of instructions past the point of the actual error, making stack traces useless. The same is generally true for null dereference exceptions, where the error often happens long after the null pointer was stored.

CAPTAIN'S LOG: NOT ENOUGH INFORMATION

On servers, the situation is somewhat better. Server applications typically generate log messages that may contain clues about why a program failed. Unfortunately, log files can be unmanageably large. Poring over logs and trying to correlate them to the source code can be extremely time consuming. Even worse, that work may yield no useful results because the logs are incomplete—that is, they simply may not provide enough information to narrow down the source of a particular error because there were not enough or the right kind of log messages.

Recent work at the University of Illinois and the University of California San Diego may lead to the development of tools that address some of these problems: SherLog[13] automates the process of tracing bugs from log messages to buggy source-code paths; and LogEnhancer[14] automatically extends log messages with information to help post-crash debugging. (More information on logging appears in a 2011 *ACM Queue* article, Advances and Challenges in Log Analysis.[11])

Despite these advances, finding bugs has actually become harder than ever. It was already challenging to find bugs when programs were sequential, but now, with multithreaded programs, asynchrony, and multiple cores, the situation has become far worse. Every execution of these *nondeterministic* programs is completely different from the last because of different timing of events and thread interleavings. This situation makes reproducing bugs impossible even with a complete log of all input events—something that would be too expensive to record in practice, anyway.

BUMPERS, SEATBELTS, AND AIRBAGS

Let's shift gears for a moment to talk about cars (we'll get back to talking about software in a minute). As an analogy for the current situation, consider when cars first came onto the scene. For years, safety was an afterthought at best. When designing new cars, the primary considerations were aesthetics and high performance (think tailfins and V-8 engines).

Eventually, traffic fatalities led legislators and car manufacturers to take safety into account. Seatbelts became required standard equipment in U.S. cars in the late 1960s, bumpers in the 1970s, and airbags in the 1980s. Modern cars incorporate a wide range of safety features, including laminated windshields, crumple zones, and antilock braking systems. It is now practically unthinkable that any company would ship a car without these essential safety features.

The software industry is in a position similar to that of the automobile industry of the 1950s, delivering software with lots of horsepower and tailfins but no safety measures of any kind. Today's software even comes complete with decorative spikes on the steering column to make sure that users will suffer if their applications crash.

DRUNK DRIVING THROUGH A MINEFIELD

The potent cocktail of manual memory management mixed with unchecked memory accesses makes C and C++ applications susceptible to a wide range of memory errors. These errors can cause programs to crash or produce incorrect results. Attackers are also frequently able to exploit these memory errors to gain unauthorized access to systems. Since the vast majority of objects accessed by applications are on the heap, heap-related errors are especially serious.

Numerous memory errors happen when programs incorrectly free objects. Dangling pointers arise when a heap object is freed while it is still live, leading to use-after-free bugs. Invalid frees happen when a program deallocates an object that was never returned by the allocator by inadvertently

freeing a stack object or an address in the middle of a heap object. Double frees occur when a heap object is deallocated multiple times without an intervening allocation. This error may at first glance seem innocuous but, in many cases, leads to heap corruption or program termination.

Other memory errors have to do with the use of allocated objects. When an object is allocated with a size that is not large enough, an out-of-bound error can occur when the memory address to be read or written lies outside the object. Out-of-bound writes are also known as buffer overflows. Uninitialized reads happen when a program reads memory that has never been initialized; in many cases, uninitialized memory contains data from previously allocated objects.

Given that the industry knows it is shipping software with bugs and that the terrain is dangerous, it might make sense to equip the product with seatbelts and airbags. The ideal would be to have both resilience and prompt corrective action for any problem that surfaces in the deployed applications.

Let's focus on C/C++/Objective-C applications—the lion's share of applications running on servers, desktops, and mobile platforms—and memory errors, the number-one headache for applications written in these languages. Safety-equipped memory allocators can play a crucial role in protecting software against crashes.

THE GARBAGE COLLECTION SAFETY NET

The first class of errors—those that happen because of the misuse of `free` or `delete`—can be remedied directly by using garbage collection. Garbage collection works by reclaiming only the objects that it allocated, eliminating invalid frees. It reclaims objects only once those objects can no longer be reached by traversing pointers from the "roots": the globals and the stack. That eliminates dangling pointer errors, since by definition there can't be any pointers to reclaimed objects. Since it naturally reclaims these objects only once, a garbage collector also eliminates double frees.

While C and C++ were not designed with garbage collection in mind, it is possible to plug in a conservative garbage collector and entirely prevent `free`-related errors. The word *conservative* here means that because the garbage collector doesn't necessarily know which values are pointers (since we are in C-land), it conservatively assumes that if a value looks like a pointer (it is in the right range and properly aligned) and acts like a pointer (it points only to valid objects), then it may be a pointer.

The Boehm-Demers-Weiser conservative garbage collector is an excellent choice for this purpose: it is reasonably fast and space efficient and can be used to replace memory allocators by configuring it to treat calls to `free` as NOPs (no operations).

Of course, garbage collection may not be suitable for all C/C++ applications. It can slow down some applications and lead to unpredictable pauses or jitter. It also may not be appropriate in resource-constrained contexts such as mobile devices, since garbage collection generally requires more memory than explicit memory management to provide the same throughput.[6] For example, conservative garbage collection in Objective-C is available on the desktop but not on iOS (iPhones or iPads).

SLIPPING THROUGH THE NET

While garbage collectors eliminate `free`-related errors, they cannot help prevent the second class of memory errors: those that have to do with the misuse of allocated objects such as buffer overflows.

Runtime systems that can find buffer overflows often impose staggeringly high overheads, making them not particularly suitable for deployed code. Tools such as Valgrind's MemCheck are

incredibly comprehensive and useful, but they are heavyweight by design and slow execution by orders of magnitude.[8]

Compiler-based approaches can substantially reduce overhead by avoiding unnecessary checks, though they entail recompiling all of an application's code, including libraries. Google has recently made available AddressSanitizer (http://code.google.com/p/address-sanitizer/wiki/AddressSanitizer), a combination of compiler and runtime technology that can find a number of bugs, including overflows and use-after-free bugs. While AddressSanitizer is much faster than Valgrind, its overhead remains relatively high (around 75 percent), making it useful primarily for testing.

All of these approaches are based on the idea that the best thing to do upon encountering an error is to abort immediately. This is the classic approach of just popping up a window indicating that something terrible has happened and would you like it to send a note home to Redmond / Cupertino, etc. This fail-stop behavior is certainly desirable in testing, but it is not usually what users want. Most application programs are not safety-critical systems, and aborting them in midstream can be an unpleasant experience for users, especially if it means they lose their work. In short, users generally would prefer that their applications be fault tolerant whenever possible.

BOHR VERSUS HEISENBERG

In fact, the exact behavior users do *not* want is for an error to happen consistently and repeatedly. In his classic 1985 article, "Why do computers stop and what can be done about it?"[5] Jim Gray drew a distinction between two kinds of bugs. The first kind are bugs that behave predictably and repeatedly—that is, they occur every time the program encounters the same inputs and goes through the same sequence of steps. These are *Bohrbugs*, named for the Bohr atom, by analogy with the classical atomic model where electrons circle around the nucleus in planetary-like orbits. Bohrbugs are great when debugging a program, since they are easier to reproduce and find their root causes.

The second kind of bug is the Heisenbug, named for Heisenberg's Uncertainty Principle and meant to connote the inherit uncertainty in quantum mechanics, which are unpredictable and cannot be reliably reproduced. The most common Heisenbugs these days are concurrency errors (a.k.a. race conditions), which depend on the order and timing of scheduling events to appear. Heisenbugs are also often sensitive to the *observer effect*; attempts to find the bug by inserting debugging code or running in a debugger often disrupt the sequence of events that led to the bug, making it go away.

Jim Gray makes the point that Bohrbugs are great for debugging, but users would rather their bugs be Heisenbugs. Why? Because Bohrbugs are showstoppers for users: every time the user does the same thing, he or she will encounter the same bug. With Heisenbugs, on the other hand, the bugs often go away when you run the program again. This is a perfect match for the way users already behave on the Web. If they go to a Web page and it fails to respond, they just click "refresh" and that usually solves the problem.

Thus, one way to make life better for users is to convert Bohrbugs into Heisenbugs—if we can figure out how to do that.

DEFENSIVE DRIVING WITH DIEHARD

My graduate students at the University of Massachusetts Amherst and I, in collaboration with my colleague Ben Zorn at Microsoft Research, have been working for the past few years on ways to protect programs from bugs. The first fruit of that research is a system called DieHard (http://diehard-software.org/) that makes memory errors less likely to impact users. DieHard eliminates some errors entirely and converts the others into (rare) Heisenbugs.

To see how DieHard works, let's go back to the car analogy. One way to make cars less likely to crash into each other is to space them farther apart, providing adequate braking distance in case something goes wrong. DieHard provides this "defensive driving" by taking over all memory-management operations and allocating objects in a space larger than required.

This de facto padding increases the odds that a small overflow will end up in unallocated space where it can do no harm. DieHard, however, doesn't just add a fixed amount of padding between objects. That would provide great protection against overflows that are small enough, and zero protection against the others. In other words, those overflows would still be Bohrbugs.

Instead, DieHard provides *probabilistic memory safety* by randomly allocating objects on the heap. DieHard adaptively sizes its heap to be a bit larger than the maximum needed by the application; the default is 1/3.[2,3] DieHard allocates memory from increasingly large chunks called *miniheaps*.

By randomly allocating objects across all the miniheaps, DieHard makes many memory overflows benign, with a probability that naturally declines as the overflow increases in size and the heap becomes full. The effect is that, in most cases when running DieHard, a small overflow is likely to have no effect.

DieHard's random allocation approach also reduces the likelihood of the `free`-related errors that garbage collection addresses. DieHard uses bitmaps, stored outside the heap, to track allocated memory. A bit set to 1 indicates that a given block is in use; 0 means it is available.

This use of bitmaps to manage memory eliminates the risk of double frees, since resetting a bit to 0 twice is the same as resetting it once. Keeping the heap metadata separate from the data in the heap makes it impossible inadvertently to corrupt the heap itself.

Most importantly, DieHard drastically reduces the risk of dangling pointer errors, which effectively go away. If the heap has 1 million freed objects, the chance that you will immediately reuse one that was just freed is literally one in a million. Contrast this with most allocators, which immediately reclaim freed objects. With DieHard, even after 10,000 reallocations, there is still a 99 percent chance that the dangled object will not be reused.

Because it performs its allocation in (amortized) constant time, DieHard can provide added safety with very little additional cost in performance. For example, using it in a browser causes no perceivable performance impact.

At Microsoft Research, tests with a variant of DieHard prevented about 30 percent of all bugs in the Microsoft Office database with no perceivable impact on performance. Beginning with Windows 7, Microsoft Windows now ships with an FTH (fault-tolerant heap) that was directly inspired by DieHard. Normally, applications use the default heap, but after a program crashes more than a certain number of times, the FTH takes over. Like DieHard, the FTH manages heap metadata separately from the heap. It also adds padding and delays allocations, though it does not provide DieHard's probabilistic fault tolerance because it does not randomize allocations or deallocations. The FTH approach is especially attractive because it acts like an airbag: effectively invisible and cost-free when everything is fine, but providing protection when needed.

EXTERMINATING THE BUGS

Tolerating bugs is one way to improve the effective quality of deployed software. It would be even better if somehow the software could not only tolerate faults but also correct them. We developed Exterminator, a follow-on system to DieHard, that does exactly that.[9,10]

Exterminator uses a version of DieHard extended to detect errors (called DieFast). While DieHard probabilistically tolerates errors, DieFast also probabilistically detects them. When Exterminator discovers an error, it dumps a *heap image* that contains the complete state of the heap without the actual contents. Exterminator then processes one or more heap images to locate the source of the error. Randomization means that each heap image will have completely shuffled objects, making it possible for Exterminator to apply a probabilistic error-isolation algorithm to identify buffer overflows and dangling pointer errors based on the resulting patterns of heap corruption. It can then compute what kind of error happened and where it occurred.

Not only does Exterminator send this information back to the programmers so they can repair the software, but it also automatically corrects the errors via *runtime patches*. For example, if it detects that a certain object was responsible for a buffer overflow of eight bytes, it will always allocate such objects (distinguished by their call site and size) with an eight-byte pad. If an object is prematurely freed, Exterminator will defer its reclamation. Exterminator can learn from the results of multiple runs or multiple users, so it could be used to proactively push out patches to prevent other users from experiencing errors it has already detected elsewhere.

THE FUTURE: SAFER, SELF-REPAIRING SOFTWARE

My group and others (notably Martin Rinard at MIT, Vikram Adve at the University of Illinois, Yuanyuan Zhou at UC-San Diego, Shan Lu at the University of Wisconsin, and Luis Ceze and Dan Grossman at the University of Washington) have made great strides in building safety systems for other classes of errors. We have recently published work on systems that prevent concurrency errors, some of which we can eliminate automatically (http://plasma.cs.umass.edu/emery/software).

Grace is a runtime system that eliminates concurrency errors for concurrent programs that use fork-join parallelism. It hijacks the threads library, converting threads to processes "under the hood," and uses virtual memory mapping and protection to enforce behavior that gives the illusion of a sequential execution, even on a multicore processor.[1] Dthreads (deterministic threads) is a full replacement for the Posix threads library that enforces deterministic execution for multithreaded code.[7] In other words, a multithreaded program running with Dthreads never has races; every execution with the same inputs generates the same outputs. Dthreads uses similar mechanisms to Grace and adds deterministic locking and diff-based resolution of conflicts, allowing it to support arbitrary multithreaded programs.

We look forward to a day in the not-too-distant future when such safer runtime systems are the norm. Just as we can now barely imagine cars without their many safety features, we are finally adopting a similar philosophy for software. Buggy software is inevitable, and when possible we should deploy safety systems that reduce their impact on users.

REFERENCES

1. Berger, E. D., Yang, T. Liu, T., Novark, G. 2009. Grace: safe multithreaded programming for C/

C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'09)*: 81–96.

2. Berger, E. D., Zorn, B. G. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*: 158–168.
3. Berger, E. D., Zorn, B. G. 2007. Efficient probabilistic memory safety. Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts, Amherst.
4. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM* 53(2): 66–75.
5. Gray, J. 1985. Why do computers stop and what can be done about it? Tandem TR-85.7; http://www.hpl.hp.com/techreports/tandem/TR-85.7.html.
6. Hertz, M., Berger, E. D. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05):* 313–326.
7. Liu, T., Curtsinger, C., Berger, E.D. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*: 327–336.
8. Nethercote, N., Seward, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of* the *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*: 89–100.
9. Novark, G., Berger, E. D., Zorn, B. G. 2007. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*: 1–11.
10. Novark, G., Berger, E. D., Zorn, B. G. 2008. Exterminator: automatically correcting memory errors with high probability. *Communications of the ACM* 51(12): 87–95.
11. Oliner, A., Ganapathi, A., Xu, W. 2011. Advances and challenges in log analysis. *ACM Queue* 9(12); http://queue.acm.org/detail.cfm?id=2082137.
12. Symantec. 2006. Internet security threat report, volume X (September); http://www.symantec.com/threatreport/archive.jsp.
13. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S. 2010. Sherlog: error diagnosis by connecting clues from runtime logs. In *Proceedings of the 15th Architectural Support for Programming Languages and Operating Systems (ASPLOS '10):* 143–154.
14. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems* 30(1): 4:1–4:28.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

EMERY BERGER is an associate professor in the department of computer science at the University of Massachusetts, Amherst, where he leads the PLASMA lab. He graduated with a Ph.D. in computer science from the University of Texas at Austin in 2002 and has spent several years as a visiting scientist at Microsoft Research and a senior researcher at the Universitat Politecnica de Catalunya/Barcelona Supercomputing Center. Berger's research spans programming languages, runtime systems, and

operating systems, with a particular focus on systems that transparently improve reliability, security, and performance. He is the creator of various widely used software systems including Hoard and DieHard. He is a senior member of ACM and an associate editor of the *ACM Transactions on Programming Languages and Systems*.