Enforcing Murphy's Law for Advance Identification of Run-time Failures*

Zach Miller
zmiller@cs.wisc.edu
University of Wisconsin–Madison

Todd Tannenbaum
tannenba@cs.wisc.edu
University of Wisconsin–Madison

Ben Liblit
liblit@cs.wisc.edu
University of Wisconsin–Madison

Abstract

Applications do not typically view the kernel as a source of bad input. However, the kernel can behave in unusual (yet permissible) ways for which applications are badly unprepared. We present *Murphy*, a language-agnostic tool that helps developers discover and isolate run-time failures in their programs by simulating difficult-to-reproduce but completely-legitimate interactions between the application and the kernel. Murphy makes it easy to enable or disable sets of kernel interactions, called *gremlins*, so developers can focus on the failure scenarios that are important to them. Gremlins are implemented using the `ptrace` interface, intercepting and potentially modifying an application's system call invocation while requiring no invasive changes to the host machine.

We show how to use Murphy in a variety of modes to find different classes of errors, present examples of the kernel interactions that are tested, and explain how to apply delta debugging techniques to isolate the code causing the failure. While our primary goal was the development of a tool to assist in new software development, we successfully demonstrate that Murphy also has the capability to find bugs in hardened, widely-deployed software.

1 Introduction

1.1 Motivation

Despite extensive in-house regression testing, buggy software is still released for a variety of reasons including incomplete test coverage, unexpected user inputs, and different run-time environments. Software developers want to systematically discover, identify, and fix application run-time failures before they affect users in the field. One challenge towards accomplishing this lofty goal is non-deterministic behavior at the level between the application and the kernel. A typical application makes thousands of calls into the kernel, and most of the time these calls respond in a repeatable manner. However, under certain run-time environment conditions, system calls into the kernel that typically succeed may return with legitimate but unexpected values.

A simple example is the `write()` system call: it usually succeeds when given valid input parameters, but fails if the disk is full. Does a given program behave in an acceptable and predictable manner in the event of a full disk? Often development teams only learn the answer when users report failures in the field. Another example is the `read()` system call, which can legitimately return fewer bytes than requested by the caller. This may happen if an interrupt occurs or if a slow device does not have all requested data immediately available. Do programs always check the number of bytes returned by a `read()` and react appropriately?

Complicating the situation is the fact that environmental conditions which bring about unexpected return values from the kernel are often hard to replicate in a typical automated testing environment. For instance, how should a regression test suite validate proper behavior in the event of a full disk? Actually filling the disk to capacity causes problems for other processes on the machine. Mounting a loopback device volume requires superuser privileges [12]. Even creating a virtual machine with a full disk may not solve the problem, as this could cause faults in the test harness itself. Other environmental conditions can be even more challenging to reproduce. The consequence is that developers fail to perform continuous integration testing under these conditions.

Across many imperfect human endeavors, Murphy's Law pessimistically predicts that "**If anything can go wrong, it will.**" Unfortunately, this does not apply when testing software. Testing would find more bugs sooner if Murphy's Law were more strictly enforced.

*Supported in part by DoE contract DE-SC0002153, LLNL contract B580360, and NSF grant CCF-0953478. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

1.2 Approach

Given the observation that a program ultimately interacts with its environment via the kernel interface, we offer a tool, called *Murphy*, to serve as an interposition agent between the application being tested and the kernel interface. Interposing at the kernel interface allows us to simulate a wide variety of environmental events. We allow enabling and disabling different sets of system call transformations, or *gremlins*, so developers can focus on the failure scenarios that are important to them. For example, when the application requests bytes from a file descriptor, the *readone* gremlin rewrites the system call to ask for and return one byte at a time.

Beyond the gremlins themselves, Murphy offers several additional mechanisms to steer its behavior. A flexible activation policy language lets developers focus gremlin activity based on the call location, values of actual arguments to the call, and various other run-time properties. A replayable gremlin activation log allows deterministic reproduction of failures and iterative root-cause analysis via delta debugging [17]. The Murphy run-time API lets programs under test dynamically steer Murphy's actions based on the program's own internal state, further supporting automated testing and debugging.

The remainder of this paper is organized as follows. Section 2 describes the architecture of Murphy, including example gremlins and run-time steering mechanisms. Section 3 provides our results running Murphy, and related work is presented in Section 4. We conclude and suggest future work in Section 5.

2 Architecture and Implementation

Our descriptions here are necessarily brief; additional details appear in a companion technical report [10].

2.1 System Call Interposition

We use a customized version of the Parrot Virtual File System tool [15] as the basis for our interposition mechanism. Parrot handles core tasks such as intercepting I/O-related system calls, decoding arguments, and replacing selected calls with new functionality. All of these actions are performed in user-space with no kernel modifications or special administrative privileges. Most uses of Parrot concern I/O virtualization for large-scale, distributed systems. We use Parrot here to simplify building gremlins.

2.1.1 Use of `ptrace`

Our implementation uses the `ptrace` interface to optionally modify interactions with the kernel, essentially acting as a "system interface interposition agent." [7] When the application under test invokes a system call, the kernel suspends that process and passes control to Murphy, which intercepts and inspects the call. At this point, Murphy may decide to tamper with the program's execution by using the `ptrace` mechanism to peek (read) or poke (write) bytes into the traced program's address space. Depending upon the system call trapped and which gremlins are configured to be active, Murphy will either

1. pass the system call to the kernel and then pass the response back to the application without any modification of the input or output arguments;
2. immediately return a failure to the application without actually passing the request to the kernel; or
3. modify the input arguments to the system call before passing the call to the kernel, and pass the actual response back to the application.

Murphy is able to track and trace entire process families by trapping `fork()`, `clone()`, `getppid()` and others, and forwarding signals and process exit codes.

2.1.2 Trade-offs of `ptrace` Interposition

Our system call interposition approach has pros and cons. There is great ubiquity by trapping via `ptrace`, and great flexibility by interposing in user-space. One major benefit is being language agnostic: Murphy works with applications written in any language, including increasingly popular managed languages such as Python and Java. No source code is required, and environmental failures can be simulated without root privileges and without impacting other processes on the system not targeted for testing. Because Murphy supports tracing entire process trees, it is possible to test software stacks (such as LAMP) consisting of many different programs written in different languages, in addition to programs that are statically linked and/or linked with C run-times other than glibc.

This approach also has challenges. The Linux system call interface does not necessarily correspond neatly to application actions. For example, all of the network socket calls are multiplexed into one (complicated) system call. Similarly, the mapping between thread creation and coordination as familiarly described by the POSIX threads API manifests itself via a strange brew of `clone()` and `futex()` system calls. When developing new gremlins, figuring out how these APIs map onto the system call interface can be a time-consuming exercise. Furthermore, recent versions of the Linux kernel introduced *vsyscall* and *vDSO* mechanisms to accelerate system calls that do not require any real level of privilege to run, such as `gettimeofday()` [1]. Calls that use these mechanisms do not cross the user/kernel boundary and therefore are invisible to Murphy.

2.2 Gremlins

We implement several example gremlins within this general framework. These include gremlins that immediately return legitimate error codes such as errno `EINTR` or errno `EAGAIN`; gremlins that modify `read()` and `write()` to simulate interrupted I/O; gremlins that introduce different amounts of latency; and special-purpose gremlins that span multiple system calls, for example simulating a full disk partition by returning errno `ENOSPC` in situations where that would make sense. In general, gremlins can further be divided into two categories: *halting* and *non-halting*. Halting gremlins typically prevent the application from making any further progress, such as *enospc* that simulates a full disk. When enabling halting gremlins, a developer can test that an application does not simply crash or abort, but instead correctly handles the situation by shutting down in an acceptable manner and reporting the error to the end-user. On the other hand, non-halting gremlins such as *readone* (causes `read()` to return one byte at a time) should not typically cause program failure. If a program's regression test suite passes with no gremlins, it should continue to pass with any non-halting gremlins activated.

Gremlins require defined composition and precedence rules. For example, both the *enospc* and the *writeone* gremlins tamper with the `write()` system call. If two or more gremlins trap the same system call, can their behaviors be combined, and if not, which one should have priority? In our current implementation, composition and precedence rules are hard-coded into Murphy.

2.2.1 Challenges to Writing Realistic Gremlins

Implementation of a single gremlin may require trapping multiple system calls. For instance, consider the *enospc* gremlin. Trapping just `write()` is not sufficient to simulate a full disk. `open()`, `mknod()`, `mkdir()`, `rename()` and over a dozen other system calls could fail due to a full disk. Murphy traps all of these.

Other gremlins may need to trap and record data from multiple system calls in order to correctly reconstruct kernel state and keep interactions legitimate. For example, consider the *cwdlongpath* gremlin that simulates executing the program with the current directory set to a very long path. At first blush, this sounds simple: just trap the `getcwd()` system call and return errno `ERANGE` if the size of the caller's buffer is smaller than the maximum allowed POSIX path length. But what if the program explicitly does a `chdir()` to `/usr`, and then invokes `getcwd()`? If `/usr` is not a symbolic link, the caller may safely assume that a smaller buffer is sufficient.

Another example is our desire for gremlins which operate on file descriptors to be conditionally activated based on the fully qualified path name referenced by the descriptor. To accomplish this, Murphy always traps `open()` to maintain mappings from file descriptors to file names. At later calls, these mappings allow "decoding" file descriptors so that they can be made available as file names for use with gremlin conditional activation (see Section 2.3). Argument decoding requires extra care for gremlins that operate on multiple system calls, as the meanings of arguments vary from one call to another. Finally, to make this useful in practice (for example, simulating `/tmp` being full), Murphy also needs to store path names that are fully qualified and canonicalized, meaning Murphy needs to track the current working directory, resolve relative paths, and expand symbolic links.

2.3 Use, Configuration, and Run-time API

To use Murphy, a developer simply invokes it with the name of the program to debug as a command-line argument. Optional command-line switches can specify the location of a configuration file and/or request the creation or replay of a gremlin activation log (see Section 2.4).

Each gremlin can be independently configured to be active, inactive, or conditionally active using a text-based configuration file. Activation conditions can be expressed in the ClassAd declarative policy language [14], providing great flexibility. A condition can be as simple as a random activation probability or can be more complex such as "activate when the file descriptor passed to this call corresponds to a file name that matches `lib*.so`."

A run-time API complements and extends static configuration. By calling into this API, the application under test can set arbitrary metadata, which is included in the gremlin activation log. Metadata might include source location information or relevant program state variables. Additional API functions allow the configuration described above to be modified dynamically for fine-grained, program-directed control over gremlin activation. Lastly, API functions allow the program to detach from Murphy and either suspend execution or immediately attach a debugger to the program under test. This helps the programmer follow their code into an area where they suspect it misbehaves. Taken together, the facilities offered by Murphy's run-time API help bridge the gap between an observed failure and the real root cause.

2.4 Reproduction of Failures

Reliably reproducing failures is essential to software testing and debugging. If Murphy is to assist developers beyond just alerting them to the existence of a bug, it must be able to reproduce the problem on demand. Note that even if a program's system call profile is deterministic, the interleaving of system calls across multiple processes is decidedly non-deterministic. In order to reproduce

gremlin-induced failures in multi-process code, we minimize non-deterministic behavior as follows:

1. Each gremlin has a separate pseudo-random number generator (PRNG) seed and state. Invoking the *readone* gremlin any number of times does not affect the PRNG for the *writeone* gremlin.

2. Multiple invocations of Murphy yield the same sequence of pseudo-random numbers.

3. For each process spawned by the application under test, Murphy maintains distinct system call statistics, gremlin states, PRNG state, and metadata.

4. Because the process ID (pid) assigned by the operating system changes during each re-run, Murphy assigns each newly spawned processes a virtual, monotonically increasing pid, or *vpid*. System call activity by this process is tracked using the tuple (pid, vpid).

Murphy can log an event whenever a gremlin modifies a system call. This log, called the *gremlin activation log*, contains a record indexed by the tuple (gremlin name, vpid) with the following fields: (1) how many times this particular gremlin was consulted to see if it wanted to modify the system call, (2) how many times Murphy has actually modified the system call, (3) the total count of all system system calls invoked by this vpid, and (4) the current value of user-supplied metadata for this process. Because this log uses the virtualized pid, and keeps track of the various system call statistics per vpid, successive runs of Murphy tracing the same program yield the same results, provided the program itself is deterministic.

Murphy can be instructed to replay the gremlin activation log while executing the program again, which produces the same results for deterministic programs. Murphy prints a warning if the count of total system calls for a given process does not match the log when a gremlin activation is replayed, letting the user know that things are not replaying identically. However, this is not fatal. In fact, it must be allowed later when minimizing the replay log (Section 2.5): removing certain gremlin invocations (such as *readone*) can affect how many subsequent system calls (such as `read()`) occur.

2.5 Fixing Failures

One disadvantage of interposing at the system-call level is a disconnect between these calls and the application developer's view of the operations being performed. This disconnect could create an understanding gap when it comes time for the developer to localize and fix errant behavior discovered by Murphy. While we assert that the mere existence of a tool that can discover such errors on a multi-process and possibly multi-language application is of value, we also support an automated strategy to help bridge this gap.

The first step is to use delta debugging [17] to shrink the failure-inducing gremlin activation log, thereby isolating just a few system calls that need to be manipulated to reproduce the failure. The second step uses Murphy to replay the minimized gremlin activation log, but now configured to suspend and detach from the application immediately upon replaying the last event in the log.

Delta debugging makes it easy for the programmer to focus their attention on the important system calls that behaved differently under Murphy. However, while very effective at minimizing gremlin activity, this does not completely bridge the gap between kernel interactions and source code. Thus, suspending after the last event leaves the program in a state where things are just about to go wrong. The user can attach with a debugger and directly observe the program's response to the manipulated system calls. In our experiments, this often results in a stack trace that pinpoints the exact line of buggy code.

3 Experimental Results

We applied Murphy to a variety of heavily-used open source packages. We ran the regression test suites of these packages primarily with non-halting gremlins enabled, and considered failed tests as bug candidates. We applied delta debugging per Section 2.5 to narrow down the code to be inspected. Often the activation log shrank to just a single system call, correlated with exactly one line of code. For example, we found a Perl interpreter bug by starting with an activation log containing 114,019 interleaved read and write system calls; delta debugging reduced this to just one vulnerable call that sufficed to cause the failure.

3.1 Ability to Detect and Pinpoint Bugs

Practically everything we tested failed with the *eagain* and *eintr* gremlins enabled. We could not run a single regression test suite with these gremlins active, as many test harnesses rely on tools like *make* that failed under the influence of these gremlins. The man pages for various system calls clearly document that they may return errnos `EAGAIN` or `EINTR`. Yet it seems that almost nothing actually checks for them. We also decided to forgo systematic testing with the latency-introducing gremlins given the limited time available for experimentation, because these obviously make the test suites run much slower.

Given the above, we focused our efforts primarily on testing with the *readone* and *writeone* gremlins. We found that even widely-used software failed to check for (or retry after) short reads or writes. Bash and Perl failed with short writes, while the widely-used OpenSSL library failed with short reads. The ubiquitous glibc also failed with short

reads: the Linux dynamic loader failed if it could not read an executable or shared library's ELF header in one `read()`. Even the trivial `/bin/true` program failed in this manner. This is a sobering sign that the problems Murphy targets are truly endemic, affecting even the most basic functionality of the system.

Problems were by no means limited to C code, or even to compiled code in general: short writes caused failures in both the Perl and Python regression test suites. The Perl and Python interpreters propagated Murphy-induced unusual behavior up into scripts. This is consistent with both interpreters' documented behavior, but the scripts themselves were unprepared for the consequences. Across a variety of application domains and languages, the methodology discussed in Section 2.5 allowed us to quickly and easily pinpoint each bug in the source code. We have reported some bugs to upstream developers and expect to report more in the future.[1]

3.2 Performance

Instrumenting a Linux process through `ptrace` incurred overhead due to the nature of trapping every system call: this requires several context switches between user and kernel space even if Murphy leaves the call unchanged. To get a feel for this overhead, we measured the wall-clock time of running the OpenSSL test suite with and without Murphy instrumentation. First we ran the test suite with no gremlins enabled. This measured the `ptrace` overhead exclusive of any repercussions from manipulating the system calls. This took 34 seconds instead of the non-Murphy baseline of 6 seconds, for a slowdown of about 5.7×. Next we ran the test suite with the non-halting gremlins enabled. This incurred significant overhead: 325 seconds, or a 54× slowdown from the baseline. This was primarily due to *readone* and *writeone*: these gremlins dramatically increased the number of system calls that are actually invoked over the lifetime of a process.

To mitigate the performance impact of adding system calls, we added stateful gremlins similar to *readone* and *writeone*, called *readone_s* and *writeone_s*. These read/write one byte on the first invocation, and also remember the count of bytes that were requested but not read/written. If the next dynamic invocation asks to transmit exactly that remainder, then this suggests that the program under test is noticing the incomplete read/write and looping accordingly. It is likely that the program will continue to do so until its original request is fully satisfied. Therefore, when we see such a compensating invocation, we pass that second call into the kernel unmolested and reset the gremlin state for the next call.

[1] `http://lists.gnu.org/archive/html/bug-bash/2012-01/msg00066.html`, `http://sourceware.org/bugzilla/show_bug.cgi?id=13601`

3.3 Validity

A bug candidate can be a false positive if Murphy simulates behavior which is truly impossible, not merely unusual. This may be due to platform-specific semantics of certain I/O devices or other POSIX mechanisms not reflected in Murphy gremlin logic. For example, Linux pipes are explicitly documented as atomic for small writes (i.e. smaller than `PIPE_BUF`). Therefore the *writeone* gremlin is overly pessimistic for writes to Linux pipes.

Another example is reading from the pseudo-random number source, `/dev/urandom`. Some specifications require that reads from this pseudo-device block until enough system entropy is available to satisfy the entire request. This is not a settled matter, however, and has been debated among highly knowledgeable developers [3]. We suggest that the mere existence of this debate argues in favor of programming defensively, regardless of what the developers may eventually decide.

Beyond validity on any one platform, one goal of Murphy is to identify problematic code before it reaches an environment in which it fails. If code is ported to a new platform, the specialized semantics of one OS may not apply. For example, an OS for embedded devices may have smaller buffers and may make weaker guarantees than our reference platform. Some platforms may not support all of POSIX.1, or may not support the most recently ratified standard. Part of Murphy's value is its ability to identify these potential problems even on a platform where such behavior is impossible. Thus Murphy is especially helpful when cross-platform portability is a goal.

4 Related Work

Our work is closely related to *software fault injection* (SFI), which traps certain calls and introduces faults [6]. Some SFI work actually corrupts memory, registers, or returned data [4]. We return rare-but-legitimate values, never corrupting an otherwise-valid system. SFI often targets only specific areas of the system, using custom device drivers [16] or operating at the boundary between shared libraries and the application [8]. Murphy traps at the `ptrace` level and can intercept all system calls, allowing a much broader range of faults to be injected.

Fuzz testing runs programs on random inputs, often triggering failures due to lax input validation [9]. However, programmers rarely view the kernel itself as potentially disruptive; our work shows the risks of this oversight by "fuzzing" the program from an unexpected direction. In addition, while fuzz inputs are often invalid, Murphy interferes in unusual but technically valid ways.

Dynamic memory-access checkers detect a narrow class of errors relating to pointer abuse [5, 11, 13]. They cannot expose errors that occur only rarely in adverse

environments unless the program is actually run in such an adverse environment. Murphy is complementary, as it creates exactly these adverse environments. Using Murphy and a memory-access checker simultaneously may reveal additional memory bugs that only manifest under the unusual circumstances that Murphy brings forth.

Many testing tools require access to the program's source code. Our approach is purely black box, suitable for robustness testing even of commercial, off-the-shelf (COTS) executables. Another black-box alternative is to add gremlins directly into an operating system, such as by modifying the User Mode Linux (UML) virtual machine [2]. However, this would not provide any more information than we can get via `ptrace`. It could also destabilize components not targeted for testing, making the entire analysis less deterministic.

5 Conclusions and Future Work

Murphy helps application developers trigger, reproduce, and diagnose bugs arising from legitimate but unexpected kernel responses. Our approach uncovers several bugs even in widely deployed and well-tested code. Given this, we anticipate this approach will be even more valuable in the hardening and testing of new software.

Clearly, additional gremlins will expose more classes of bugs. Gremlins that simulate temporary network problems may be especially fruitful. Richer information about system call contexts will allow finer-grained gremlin activation and thus a more targeted hunt for some specific bugs. Additional state tracking by existing gremlins may reduce false positives and improve performance.

Murphy reveals pervasive bugs even in well-tested programs. Nothing we tested handled errnos `EINTR` or `EAGAIN` without failure. This raises an important question: is it naïve for a kernel to return these responses if nothing is going to deal with them correctly? Perhaps instead these types of failures should be squashed in the OS or run-time libraries before returning to the application. Our experience shows that this may be the only practical means to ensure these cases are handled correctly.

We wish to explore the creation of a defensive software-hardening tool that squashes exactly the sort of errors that Murphy simulates. So many programs seem to have problems correctly handling various responses, especially errnos `EAGAIN` and `EINTR`. Therefore perhaps there is a need for such a hardening tool, complete with its own policy language describing how to handle errors (retry, block until success, timeout, no change, etc.). A more comprehensive survey of existing applications' behavior under Murphy would improve our understanding of software's implicit assumptions. This may motivate further research on mitigation strategies.

References

[1] J. Corbet. On vsyscalls and the vDSO. `http://lwn.net/Articles/446528/`, June 2011.

[2] J. Dike. *User Mode Linux*. Prentice Hall, Upper Saddle River, NJ, 2006. ISBN 0131865056.

[3] U. Drepper. short read from `/dev/urandom`. `https://lkml.org/lkml/2005/1/13/485`, Jan. 2005.

[4] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. *Computer Performance and Dependability Symposium, International*, 0:0204, 1995. ISSN 1087-2191. doi:10.1109/IPDS.1995.395831.

[5] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[6] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, Apr. 1997. ISSN 0018-9162. doi:10.1109/2.585157.

[7] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[8] P. D. Marinescu and G. C. LFI: A practical and general library-level fault injector. In *Proceedings of the Intl. Conference on Dependeable Systems and Networks (DSN)*, Lisbon, Portugal, June 2009.

[9] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. In *In Proceedings of the Workshop of Parallel and Distributed Debugging*, pages ix–xxi. Academic Medicine, 1990.

[10] Z. Miller, T. Tannenbaum, and B. Liblit. Murphy: An environment for advance identification of run-time failures. Technical Report 1770, Department of Computer Sciences, University of Wisconsin–Madison, Apr. 2012.

[11] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV)*, 2003.

[12] P. Nguyen. Loopback devices in linux. `http://csulb.pnguyen.net/loopbackDev.html`, Apr. 2010.

[13] B. Perens. Electric Fence. `http://perens.com/FreeSoftware/ElectricFence/`, Sept. 2010.

[14] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[15] D. Thain and M. Livny. Parrot: An application environment for data-intensive computing. *Journal of Parallel and Distributed Computing Practices*, 2004.

[16] T. Tsai and R. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *Lecture Notes in Computer Science*, pages 26–40. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-60300-9. doi:10.1007/BFb0024305.

[17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28:2002, 2002.