Murphy: An Environment 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

April 27, 2012

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

*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.

testing software. Testing would find more bugs sooner if Murphy's Law were more strictly enforced.

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 [18]. 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 details the implementation of Murphy, including gremlins currently implemented and additional run-time steering mechanisms. Section 3 provides our results running Murphy, and related work is presented in Section 4. We wrap up with potential future work in Section 5 and contributions in Section 6.

2 Architecture and Implementation

2.1 System Call Interposition

We use a customized version of the Parrot Virtual File System tool [16] 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 Parrot-based Murphy implementation uses the Linux `ptrace` interface to optionally modify interactions with

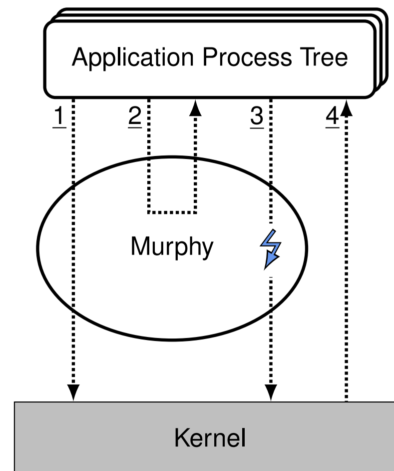

Figure 1: **System call control flow.** *Murphy traps a system call from the application and either passes it to the kernel unchanged (label 1), directly returns a given result without passing to the kernel (label 2), or passes it to the kernel after changing input arguments (label 3). Murphy currently does not modify any results coming back from the kernel (label 4).*

the kernel. When the application under test invokes a system call, the kernel suspends that process and passes control to Murphy. Murphy intercepts and inspects the call. At this point, Murphy may decide to tamper with the program's execution via the ptrace mechanism to peek (read) or poke (write) bytes into the traced program's address space – see Figure 1. Depending upon the system call trapped and which gremlins are configured to be active, Murphy does one of the following:

- 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, per labels 1 and 4 in Figure 1.
- Immediately return a failure to the application without ever actually passing the request to the kernel, per label 2 in Figure 1. This is accomplished by changing the actual system call requested to a different call without any side-effects (e.g. `getpid()`) and then modifying the response from this call to the desired error code before returning control to the application.
- Modify the input arguments to the system call before passing the call to the kernel, and then pass the actual response back to the application, per labels 3 and 4 in Figure 1.

Even if Murphy alters the response, it only does so such that the response is still legitimate under the terms of the documented call semantics or API.

The system calls `fork()` and `clone()` are always trapped to enable tracing of an entire family of processes. This way, when a process being traced forks, Murphy can invoke the necessary ptrace calls to trap all the system calls of the child process as well. A side effect of this method is that the newly-forked process is a child of Murphy and not the process that called `fork()`, and in order to hide this artifact from the application Murphy must: (1) intercept `getppid()` in order to return the correct parent pid from the application's point of view, (2) reap and store children's exit statuses upon receipt of a `SIGCHLD` signal, (3) forward `SIGCHLD` to the proper application process, and (4) intercept `wait4()` and correctly forward stored exit status codes to the application.

2.1.2 Trade-offs of ptrace Interposition

Our system call interposition approach has pros and cons. One major benefit is that it is 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 special root privileges and without impacting other processes on the system not targeted for testing. Because Murphy supports tracing an entire process tree, it is appropriate to test an entire software stack consisting of many different processes perhaps written in different languages, such as the LAMP stack. By using ptrace instead of dynamic linker `LD_PRELOAD` approaches [6], all programs can be tested, including those that are statically linked and/or linked with C runtimes other than glibc. If a program is linked with glibc, the ptrace approach enables Murphy to find problems within glibc as well. Finally, trapping via ptrace removes sensitivity to frequent glibc updates.

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]. The kernel allows the page containing the current time to be mapped read-only into user space. That page also contains a fast `gettimeofday()` implementation that does not require crossing the user/kernel boundary and is therefore invisible to Murphy. While vsyscalls can be disabled [1], doing so requires root access.

2.2 Gremlins

Table 1 presents the initial set of gremlins implemented for this project. In general, they can be divided into two categories: *halting* and *non-halting* gremlins. Halting gremlins typically prohibit the application from making any further progress. When enabling halting gremlins, such as *enospc* that simulates a full disk, 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 propagating the error to the end-user. On the other hand, non-halting gremlins such as *readone* (causes all `read()` calls to return one-byte at a time) should not typically cause program failure. If a program's regression test suite passes without any gremlins, it should continue to pass with any non-halting gremlins activated.

Gremlins require defined composition and order of precedence rules. For example, both the *enospc* and the *writeone* gremlins operate upon 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

Many gremlins seem simple to develop at the outset, but become more complicated as you dig down. 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, and in our Murphy implementation we identified and trapped all of them.

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. At first blush, it sounds simple: just trap the `getcwd()` system call and return errno `ERANGE` if the size of the caller's buffer is smaller than the max allowed POSIX pathname. But what if the program explicitly does a `chdir()` to `/usr`, and then invokes `getcwd()`? If `/usr` is not a symlink, the caller could perhaps assume it can use a smaller buffer.

Another example is our desire for gremlins which operate on file descriptors to be conditionally activated based on the fully qualified pathname referenced by the descriptor. To accomplish this, Murphy always traps `open()` to maintain mappings from file descriptors to file names. At

[1] `echo 0 > /proc/sys/kernel/vsyscall64`

Name	Severity	Implementation	Description
closefail	halting	Causes `close()` to fail with errno `EIO`.	Errors from a previous `write()` may not be reported until `close()` is called. According to the `close()` man page, "not checking the return value of close is a common but nevertheless serious programming error."
cwdlongpath	non-halting	If the program has already called `chdir()` or `fchdir()` then this gremlin does nothing. But if the program attempts to `getcwd()` without setting it first, this gremlin returns errno `ERANGE` and remembers the size of the buffer given. The buffer must be increased by at least one byte for successive calls to succeed.	This gremlin simulates executing the program with the current directory set to a very long path.
eagain	halting	Returns errno `EAGAIN` for system calls in which that is legal.	This gremlin simulates a system which is highly loaded and/or is resource constrained. For example, `fork()` returns this if (1) it is unable to copy page tables, (2) it cannot allocate an internal kernel task structure for the child, or (3) rlimit was reached.
eintr	halting	Returns errno `EINTR` for all system calls in which that is legal.	This gremlin simulates a system call which has been interrupted by a signal. Some signals cannot be blocked, and therefore this is valid regardless of the signal mask and all programs should handle this.
enospc	halting	Returns errno `ENOSPC` for all system calls in which that is legal.	This gremlin simulates a full disk partition.
readone	non-halting	Rewrites `read()` to request one byte	These three gremlins simulate a `read()` system call that was (1) interrupted by a signal, (2) reading from a pipe, FIFO, socket, or other special file that had fewer bytes than requested immediately available, or (3) reading from a file that exists on a file system that may eagerly return ready blocks (like a network file system).
readone_s	non-halting	Similar to *readone*, rewrites `read()`, but keeps state between system calls. The first time the gremlin is invoked, it returns one byte and stores the remainder of bytes requested but not returned. If the next call to `read()` requests exactly the remainder, we allow it to occur unmolested. Otherwise, Murphy again returns only one byte and stores the new remainder.	
readless	non-halting	Rewrites `read()` to return half as much data as requested (minimum one byte).	
sleepy1 *sleepy10th* *sleepy100th* *sleepy1000th*	non-halting	Add latency to any system call, ranging from one second to one thousandth of a second.	This family of gremlins simulates a highly-loaded system, and is meant to expose bugs where specific timings or races between separate processes are assumed to have predictable results.
writeone	non-halting	Rewrites `write()` to transmit one byte.	These three gremlins simulate a `write()` system call that (1) was interrupted by a signal, (2) is writing into a pipe, FIFO, or other special file that does not have enough buffer space available, (3) has caused the disk to become full, or (4) reached the maximum file size.
writeone_s	non-halting	Similar to *readone_s*, it keeps state between system calls and allows `write()` to pass through unmolested if the byte count is exactly equal to the number of bytes not written on the previous invocation.	
writeless	non-halting	Rewrites `write()` to transmit half as much data as requested (minimum one byte).	

Table 1: **Initial gremlins implemented in Murphy**

```
[
    Gremlin = "readone"
    SyscallCount = 20
    InvokedCount = 1
    EvilCount = 0
    Pid = 24068
    VirtualPid = 1
    Meta = ""
    SyscallNum = 0
    SyscallName = "read"
    FD = 3
    Name = "/lib64/libc.so.6"
    Length = 832
]
```

Figure 2: **Sample gremlin ClassAd context**

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 Usage and Policy Configuration

To use Murphy, a developer simply invokes it with the name of the program to debug as a command-line argument. Command-line switches can optionally 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, active only a specific random percent of the time, or active based upon a declarative policy language constraint. The general format of a Murphy configuration file is one line per gremlin with the following fields:

```
gremlin:percentage:seed:constraint
```

The `gremlin` field is the name of the gremlin being configured. The `percentage` field is a number between 0 and 100 that represents a random percentage the gremlin should be active (i.e. when to alter interaction with the kernel). The percentages `0` (always inactive) and `100` (always active) are special cases such that the pseudo-random number generator (PRNG) is not actually invoked. The optional `seed` field allows the developer to explicitly choose a seed for the PRNG for this gremlin.

The `constraint` field is an optional Boolean expression that must evaluate to true in order for a gremlin to activate. The expression is written in the language of ClassAds [14]. A ClassAd is a mapping from attribute names to values. A ClassAd Boolean expression uses a syntax reminiscent of C or Java, and may reference any of the named properties in the ClassAd as parameters to a rich set of functions and operators. Conditionals, string operations, and regular expression pattern matching are all supported [15]. Whenever a gremlin could potentially do mischief, Murphy creates a ClassAd context like the one in Figure 2 that contains state information regarding the gremlin and the system call parameters. The integration of a policy language like ClassAds into Murphy enables the developer to conditionally enable gremlins
based upon, for instance, how many times a call has been trapped, and/or a pattern match upon the input parameters to the kernel call being trapped. For example, the configuration line

```
readone:100:0:!regexp("(\\.so(\\.[0-9]+)*)$",Name,"")
```

uses regular expression pattern matching on the Name attribute to prevent the *readone* gremlin from activating upon files that look like a shared library.

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 2.6): removing certain gremlin invocations (such as *readone*) can affect how many subsequent system calls (such as `read()`) occur.

2.5 Control Functions

To facilitate practical usage of Murphy, we provide several Murphy "control functions" that the application being debugged can invoke to help bridge the gap between a gremlin causing a failure and finding the code actually responsible for the failure. The functions currently implemented are:

Set metadata. Metadata supplied in the high-level source language is placed into the gremlin activation log. A programmer can of course manually add calls to update the metadata at function entry or surrounding some suspicious code that is under investigation. Alternatively, a programmer could automatically wrap specific function calls that are being investigated by using preprocessor macros or other techniques to insert filename and line numbers into the metadata. Since the metadata is free-form, even the values of variables could be tracked if desired.

Update configuration. To further help debug their code, programmers can dynamically reconfigure gremlins. This can be used to (1) reseed the PRNG, (2) narrow the scope of gremlin activity by enabling and disabling certain gremlins, or (3) change a gremlin's activation constraints in order to improve the signal-to-noise ratio of the Murphy output.

Suspend and detach (with gdb). These control functions aid in live debugging of a suspected bug. When invoked, Murphy suspends the process being traced and detaches from the ptrace interface, allowing the programmer to attach to the process themselves with a debugger. Because gdb is so prevalent, we also provide a control function for convenience which automatically attaches with gdb. These mechanisms allow the programmer to follow their code into an area where they suspect it misbehaves.

2.5.1 Interface to Control Functions

Originally, we assumed programs could use `ioctl()` to trap into the Murphy controller and perform run-time control functions. However, we find that many languages do not pass `ioctl()` on directly to the kernel, or do not implement it at all. For example, in Java it requires using the Java Native Interface. As such, we settled on using `mkdir()` which is much more widely supported.

With `mkdir()`, we implement a filesystem interface similar to the `/proc` interface on Linux. If the filename starts with "`/Murphy/`" followed by the desired control function name, Murphy intercepts the `mkdir()` and instead passes it to the Murphy controller subsystem. Example invocations of the Murphy control functions include:

`mkdir /Murphy/set-metadata/foo.c:115`
(set metadata to "`foo.c:115`")

`mkdir /Murphy/update-config/readone:0`
(update configuration to disable *readone* gremlin)

`mkdir /Murphy/suspend`
(suspend and detach)

`mkdir /Murphy/suspend-and-debug`
(suspend and reattach with gdb)

2.6 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 [18] 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.

Configuration	Time
No Murphy	6 seconds
Murphy with no gremlins	34 seconds
Murphy with non-halting gremlins	325 seconds

Table 3: **Performance impact on OpenSSL test suite**

3 Experimental Results

3.1 Methodology

In order to evaluate the utility and effectiveness of Murphy, we apply it to a variety of heavily-used open source packages. We run the regression test suites of these packages primarily with non-halting gremlins enabled, and looked at failed tests as candidates for potential bugs. If bug candidates are found, we apply delta debugging to the gremlin activation log in order to minimize the number of system calls with which Murphy interfered, thereby reducing the amount of code that needed to be inspected. Often the activation log shrinks down to just a single system call that correlates to exactly one line of code. For example, the perl interpreter bug was found by starting with an activation log containing 114,019 interleaved read and write systemcalls which was delta-debugged down to just one.

Packages we test include some with source code written in languages other than C, in particular the Perl and Python regression test suites. We also experiment with using Murphy in conjunction with glibc, Bash, OpenSSL, and gcc.

3.2 Ability to Detect and Pinpoint Bugs

We discovered our first bug early in our development process: a defect in the Linux dynamic loader which is used by all dynamically-linked executables. If the dynamic loader cannot load the Linux executable ELF header in one `read()`, it fails. Even the trivial `/bin/true` program fails 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. In fact, we actually needed to work around this obstacle before other meaningful results could be gathered, which helped guide our decision to implement the policy language described in Section 2.3.

Practically everything we tested fails when the *eagain* and *eintr* gremlins are enabled. We cannot even run a single regression test suite with these gremlins active, as many test harnesses rely on tools like *make* that fail under the influence of these gremlins. It seems that despite the man page for various system calls documenting that they may return errno `EAGAIN` or errno `EINTR`, almost nothing actually checks for them. We also decided to forgo systematic testing with the *sleepyX* gremlin given the limited time available to us for experimentation, because it obviously makes the test suites run much slower.

Given the above, we focus our efforts primarily on testing with the *readone* and *writeone* gremlins, and find that even widely-deployed software has bugs involving not checking for or not retrying after a short read or write occurs. We find bugs of this class in the Perl and Python test suites, the Perl interpreter, glibc, OpenSSL, and Bash. Table 2 summarizes these findings. In addition to detecting the existence of a problem (evidenced by having regression test suites fail under the influence of gremlins), the methodology discussed in Section 2.6 allows us to quickly and easily pinpoint each bug in the source code. We have submitted bug reports to upstream developers and plan to submit more in the future. [2] [3]

3.3 Performance

Instrumenting a process through the ptrace interface on Linux incurs overhead due to the nature of trapping every system call, whether it is interfered with or not, which involves several context switches between user and kernel space. To get a feel for this overhead, we measure the wall-clock time of running the OpenSSL test suite with and without instrumentation by Murphy. First we run the test suite with no gremlins enabled, thus measuring the overhead of the instrumentation itself and not of the repercussions resulting from manipulating the system calls. Next we run the test suite again with the non-halting gremlins enabled, which may incur significant overhead primarily due to the fact that the *readone* and *writeone* gremlins can dramatically increase the number of system calls that are actually invoked over the lifetime of a process. Timing results for running the OpenSSL test suite are provided in Table 3.

To mitigate the large increase in the number of system calls and the resulting performance decrease, we added stateful gremlins for *readone* and *writeone*, called *readone_s* and *writeone_s* respectively, that read/write one byte on the first invocation but remember the count of bytes that were requested but not read/written. If the next invo-

[2] `http://sourceware.org/bugzilla/show_bug.cgi?id=13601`

[3] `http://lists.gnu.org/archive/html/bug-bash/2012-01/msg00066.html`

Package	Language	Gremlin	Description
glibc	C	*readone*	Fails to load dynamic libraries when it cannot read the entire ELF header (832 bytes) from a shared library file in one system call
Perl interpreter	C	*writeone*	When spawning a new process fails in Perl's popen implementation, child cannot write the error code (4 bytes) from `exec()` in one system call onto a pipe read by the parent, and as a result the parent panics when all 4 bytes cannot be read
Perl test suite	Perl	*writeone*	Regression test `through.t` fails due to not checking the return code from a call to Perl's syswrite subroutine.
Bash interpreter	C	*writeone*	Fails during a file redirection operation when it cannot write all bytes into a temporary file in one system call.
Python test suite	Python	*writeone*	Regression test `test_cmd_line.py` fails testing Python's Popen routine when it cannot write all bytes to `stdin` in one system call
OpenSSL library	C	*readone*	Fails when it cannot read all requested bytes from `/dev/urandom` in one system call

Table 2: **Bugs found with *readone* and *writeone* gremlins.** *Murphy both detected bugs and successfully pinpointed the bug at the source code level when monitoring the test suites of some popular software packages.*

cation asks to transmit exactly that remainder, then this is a strong sign 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 a second invocation that is compensating in this manner, we allow that second call to pass into the kernel unmolested.

3.4 Validity

After identifying bug candidates, some manual investigation is often required to confirm the existence of an actual bug. In some cases, semantics of specific I/O devices or other POSIX mechanisms may render the bug candidate innocuous as other conditions prevent this specific instance from occurring. For example, the man page on Linux for `pipe()` specifies that small writes (i.e. smaller than `PIPE_BUF`) must be atomic. This may mean that Murphy writing only one byte is not valid behavior on our reference platform if the device involved is a pipe.

Another example is reading from `/dev/urandom`, the pseudo-random number source. Some specifications declare that `read()` will block when reading from `/dev/urandom` until enough system entropy is available to satisfy the entire `read()` request. This is not a settled matter, however, and has been debated among highly knowledgeable developers, including Ulrich Drepper (lead contributor and maintainer of glibc) [2]. We suggest that the mere existence of this debate argues in favor of programming defensively, regardless of what Drepper and others may eventually decide.

Regardless of validity on any individual platform, one goal of Murphy is to help identify potentially problematic code before it reaches an environment in which it fails. If an application is ported to another platform, the specialized semantics of one OS may not apply. For example, it is conceivable that an OS designed to run on embedded devices may have much smaller internal buffers and may not make the same guarantees as our reference platform. Some target platforms may not claim to support all of POSIX.1, or may not support the most recently ratified standard.

As such, the ability of Murphy to help identify these potential problems, even on a platform where such behavior is impossible, is part of the value provided by the Murphy tool.

4 Related Work

This work is related to the concept of fuzz testing [9, 10]. Normally, fuzzing is done by providing a stream of random inputs to the program and checking whether it crashes or hangs. Because the source of the inputs is normally user-provided, from a configuration file or environment variable, or received over the network, programs should be validating these inputs and often already are (or intend to). However, applications do not typically view the kernel as potentially disruptive. Consequently, they are sometimes lax about validating the data returned by a system call, providing a different class of potential failure. In our project, the "fuzz" comes from the opposite direction. In addition, fuzz testing often produces invalid inputs, while Murphy only introduces injections that are semantically legitimate.

Existing tools like eFence [13], valgrind [11], and Pu-

rify [4] can aid in run-time detection of errors, but just for a very narrow class of error which is related to memory access. While these tools are useful for exposing programmer error in the normal operation of a program, they do not necessarily help expose errors that occur only rarely in adverse environments unless the program is actually run in such an adverse environment. Murphy creates exactly these adverse environments, and thus could be a powerful compliment to dynamic memory-safety checkers: by running a program under both Murphy and a memory-access checker simultaneously, we may discover additional memory-access bugs that only manifest under the unusual circumstances that Murphy brings forth.

This project is most closely related to techniques of previous work in *software fault injection* (SFI), which are techniques that trap certain calls and introduce faults [5]. Some of these techniques actually corrupt the data being returned, the memory, or the registers as an extreme form of fuzz testing [3]. We return values that are completely legitimate during rare circumstances while not corrupting an otherwise valid system. Many of the SFI techniques target only specific areas of the system and are implemented by writing a device driver [17], while others operate at the boundary between shared libraries and the application [7]. In comparison, Murphy traps at the ptrace level and is able to intercept all system calls, allowing a much broader range of types of faults to be injected.

User Mode Linux (UML) provides a useful virtualized environment for testing modifications to the kernel itself. One approach we could have taken would be to actually modify the kernel to behave maliciously, rather than trap the calls and manipulate the data going to and from the kernel. Ultimately, this would not provide Murphy any more information about the application than we can get via the ptrace interface, and could introduce instability in places we are not targeting for testing, potentially making the analysis less deterministic.

Many, but not all, of the tools in these fields require access to the program's object code or source code to be of use, while our approach is a purely black-box testing system.

5 Future Work

Clearly, Murphy will expose more classes of bugs with the implementation of additional gremlins. Especially useful would be additional gremlins to simulate environmental failures, such as gremlins that cause temporary network problems. Enriching the information in the system call context will allow more fine-grained triggering of gremlin activation and thus a more targeted hunt for some specific bugs. Enhancing existing gremlins with more state could provide fewer false positive bug candidates as well as reduce performance overhead.

To continue to improve Murphy and its utility to programmers, we intend to build the next version of Murphy on top of the *strace* [8] tool instead of Parrot. This would allow for better interpretation of system call arguments, better coverage of system calls, and implementation of more varied gremlins. Additionally, this will enhance the portability of Murphy to other platforms.

We would like to explore the creation of a defensive software-hardening tool that squashes exactly the types of errors that Murphy simulates. So many programs seem to have problems correctly handling various responses, particularly errno `EAGAIN` and errno `EINTR`. Therefore perhaps there is a need for such a hardening tool, complete with its own policy language describing how to handle errors (e.g. retries, block until success, timeouts, no change, etc.).

To that end, we feel it would be useful to do a more comprehensive survey of running Murphy on additional existing software and classify results by gremlin to better understand the implicit assumptions of software in the wild. This may motivate further research on mitigation strategies.

6 Conclusion

We have identified system calls which typically behave in a predictable manner under predictable environmental conditions, but can return unexpected results under conditions that can be difficult to reproduce while testing. Using these sets of system calls we developed a tool, which we named Murphy, applicable for use in typical automated testing environments which helps application developers pro-actively discover bugs resulting from failure to handle legitimate but unexpected kernel responses. Along the way, we devised an automated process to reproduce failures, identify specific gremlins responsible for the failures, and map system failures back to the errant source code

Additionally, we identified some classes of bugs that are pervasive, even in well-tested programs. Nothing we tested handled errno `EINTR` or errno `EAGAIN` without failure. This raises an important question: is it wise 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 before returning to the application, as this may be the only practical means to ensure they are handled correctly.

Finally, we observed that the approach taken by Murphy uncovered 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.

References

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

[2] Ulrich Drepper. short read from `/dev/urandom`. `https://lkml.org/lkml/2005/1/13/485`, January 2005.

[3] Seungjae Han, Kang G. Shin, and Harold 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.

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

[5] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, April 1997. ISSN 0018-9162. doi:10.1109/2.585157.

[6] Michael Kerrisk. Linux Programmer's Manual. `http://www.kernel.org/doc/man-pages/online/pages/man8/ld-linux.so.8.html`, January 2009.

[7] Paul D. Marinescu and George 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.

[8] Roland McGrath. Strace project homepage. `http://sourceforge.net/projects/strace/`, October 2011.

[9] Barton P. Miller, Lars Fredriksen, and Bryan 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] Barton P. Miller, David Koski, Cjin Pheow, Lee Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report 1268, Department of Computer Sciences, University of Wisconsin–Madison, April 1995.

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

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

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

[14] Rajesh Raman, Miron Livny, and Marvin 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] Marvin Solomon. The ClassAd language reference manual. `http://research.cs.wisc.edu/condor/classad/refman/`, May 2004.

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

[17] Timothy Tsai and Ravishankar Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In Heinz Beilner and Falko 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.

[18] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28:2002, 2002.