An Empirical Study of the Robustness of MacOS Applications Using Random Testing

Barton P. Miller Gregory Cooksey Fredrick Moore
{bart,cooksey,fredrick}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685 USA

Abstract

We report on the fourth in a series of studies on the reliability of application programs in the face of random input. Over the previous 15 years, we have studied the reliability of UNIX command line and X-Window based (GUI) applications and Windows applications. In this study, we apply our fuzz testing techniques to applications running on the Mac OS X operating system. We continue to use a simple, or even simplistic technique: unstructured black-box random testing, considering a failure to be a crash or hang. As in the previous three studies, the technique is crude but seems to be effective in locating bugs in real programs.

We tested the reliability of 135 command-line UNIX utilities and thirty graphical applications on Mac OS X by feeding random input to each. We report on application failures – crashes (dumps core) or hangs (loops indefinitely) – and, where source code is available, we identify the causes of these failures and categorize them.

Our testing crashed only 7% of the command-line utilities, a considerably lower rate of failure than observed in almost all cases of previous studies. We found the GUI-based applications to be less reliable: of the thirty that we tested, only eight did not crash or hang. Twenty others crashed, and two hung. These GUI results were noticeably worse than either of the previous Windows (Win32) or UNIX (X-Windows) studies.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Testing Tools

General Terms: Reliability

Key Words: fuzz, random testing

1 INTRODUCTION

In 1990 [15], we published our first study of the reliability of UNIX command line applications. This study was motivated by an experience "one dark and stormy night": One of the authors was connected to his office computer via a dial-up line and there was a typical midwest thunderstorm in progress. Due to the storm, there was a significant amount of noise on the phone line (the modem predated the general use of error correction). As many programmers of the day had experienced, it was a race to type a sensible command before the noise overwhelmed his typing. While the presence of noise was not surprising, the fact that the noise seemed to be causing important and commonly used utilities to crash *was* surprising. We set about to study this phenomenon systematically.

We developed *fuzz* testing, the sending of unstructured random input to an application program. With a few simple tools, we tested more than 80 command line utility programs on six versions of UNIX. As a result of this testing, we were able to crash a surprising (to us) number of programs: 25-33%. These crashes were typically caused by the use of risky programming practices that are well known to experienced programmers and the software engineering community.

In 1995 [14], we re-tested UNIX command line utilities, increasing the number of utilities and UNIX versions tested, and also extending fuzz testing to X-Window GUI applications, the X-Window server itself, network services, and even the standard library interface. Of the commercial systems that we tested, we were still able to crash 15-43% of the command line utilities, but only 6% of the open-source GNU utilities and 9% of the utilities distributed with Linux. The causes of these crashes were similar (or occasionally identical) to the 1990 study. Of the X-Window applications that we tested, we could crash or hang 26% of them based on random valid keyboard and mouse events. The causes of the crashes and hangs were similar to those of the command line utilities. The most memorable result of the 1995 study was the distinctly better reliability (under our testing) of the open-source tools.

In 2000 [5], we shifted our focus to the commodity desktop operating system, Microsoft Windows. Using the Win32 interface, we sent random valid mouse and keyboard events to the application programs and could crash or hang at least 45% of the programs tested on Windows NT 4.0 and Windows 2000.

We are back again, this time testing a relatively new and popular computing platform, Apple's Mac OS X. Mac OS X was a major step for Apple, switching to a UNIX-based (BSD) operating system with NeXTSTEP (now called "Cocoa") [2] and Apple extensions. We tested both the UNIX command line utilities and GUI-based application programs.

When starting this study, we expected the command line utilities to have excellent reliability in the context of fuzz testing. Our

RT'06, July 20, 2006, Portland, ME, USA

expectation was based on the years of published studies, the widening usage of fuzz testing, and freely available fuzz tools. While the results were reasonable, we were disappointed to find that the reliability was no better than that of the Linux/GNU tools tested in 1995. We were less sure what to expect when testing the GUI-based applications; the results turned out worse than we expected.

Specifically we found the following key results:

❏ Of the 135 command line utilities that we tested, ten crashed (a failure rate of 7%) and none hung. These results are similar to the best results (the GNU utilities) of the 1995 study.

❏ Testing the GUI-based utilities on valid mouse and keyboard input produced a large number of failures. Of the thirty programs that we tested, 20 crashed and two hung (a failure rate of 73%). This result is the worst showing that we have had in the history of our testing effort.

❏ The types of simple programming errors that led to many of the failures in 1990, 1995, and 2000 are still present in the current tests. In fact, some of the same failures found in earlier tests are still present in our new study.

The next section briefly describes the fuzz tools used in this study. The basic command line tools (fuzz and ptyjig) are little changed from their earlier form. We describe in more detail fuzz-aqua, our tool for testing the GUI-based applications. Section 3 describes our experimental methods and Section 4 provides the details of our test results. In Section 5, we analyze the results and discuss the causes of the various failures. We also provide commentary on fuzz testing, attempting to place it in context. Section 6 discusses related work and we conclude in Section 7.

2 The Testing Tools for Mac OS X

We first describe the tools that we used to test the command line utilities under Mac OS X (Section 2.1). These tools are essentially the same as those used in previous studies. We then describe the tools used to test GUI applications on Mac OS X under their Aqua interface (Section 2.2). The goal of these tools is to provide a source of random keyboard and mouse events, much as was done previously on X-Windows under UNIX [14] and Win32 under Windows [5].

2.1 The Command Line Fuzz Tools

The first part of our testing effort was to repeat the tests from the previous studies on the command-line utilities provided with Mac OS X. All the utilities we tested are included with OS X's BSD subsystem or developer-tools packages, both of which are included with the operating system (although they are not necessarily required).

To perform these tests, we used the same tools, fuzz and ptyjig, as were used in the 1990, 1995, and 2001 studies. Getting the tools to run on OS X required some minor porting (mostly updating them to ANSI C), but required no substantive changes.

The main tool used in the original studies was fuzz, a random-character generator with options to adjust basic characteristics of its output. The options that we used were:

- -0: Whether null (zero-byte) characters are included in the output; null bytes often confuse string processing.
- -p: Whether to include only printable ASCII characters or all characters; non-printable characters can cause sign-bit problems in variables of type `char`.
- A number with which to seed the random-number generator.
- The number of characters to produce.

By varying these options, we generated 24 files of random characters: one for each combination of two different random seeds, three different file sizes (1,000, 10,000, or 100,000 characters), -0 and -p. These are similar to the option combinations that we used in previous studies. Once the files were generated, it was a simple matter to provide each file as input to our target applications.

For a small number of applications that require access to the underlying terminal device, simply piping characters to the application does not work. This problem is solved by ptyjig, which will run an arbitrary command-line application in a pseudo-terminal, sending its standard input to the target application in an acceptable manner. Normal fuzz data can then be piped to ptyjig and will be presented to the application as desired.

The applications for which we needed ptyjig were less, emacs, nano, and vim.

2.2 The GUI Fuzz Tools

The second part of our testing was to evaluate the reliability of applications that use graphical user interfaces. Most new, significant applications are based on such interfaces, so this type of testing is crucial for completeness. As in the previous Win32 and X-Windows studies, we developed techniques to send valid keyboard and mouse inputs to the target applications. The previous studies also generated invalid inputs, i.e., those that could never be generated from the keyboard or mouse, but we chose not to repeat those studies as they produce few useful insights.

Such a task requires that we send manufactured events to an application such that the events are indistinguishable from normal user input. Figure 1 shows the path that a user input event takes as it makes its way through the Mac OS X system from an input device to the active application. The event arrives from the device drivers at the window server, where it is forwarded to the correct application's event queue based on which application is currently active and the current position of the mouse on the screen. Mac OS X provides four points in the event path, called event *taps*, where a program can insert or eavesdrop on events passing through the system. The first tap is the point at which device drivers insert the events that they have created from physical device I/O into the system. The second is the point where those device-created events and remote operation events enter a user session. At the third tap, the events have been annotated by the window server as intended for a specific application. The fourth tap is where the window server sends events to applications' event queues.

We had intended to insert all of our synthesized events into this last event tap so that we could be sure that key presses and mouse clicks were only sent to the application that we were testing. Unfortunately, the system API that is available for creating and inserting user input events into the event stream does not function properly for mouse events, so we used the remote operation API to create and send mouse events to our target application. Figure 1 shows the routes by which we send events.

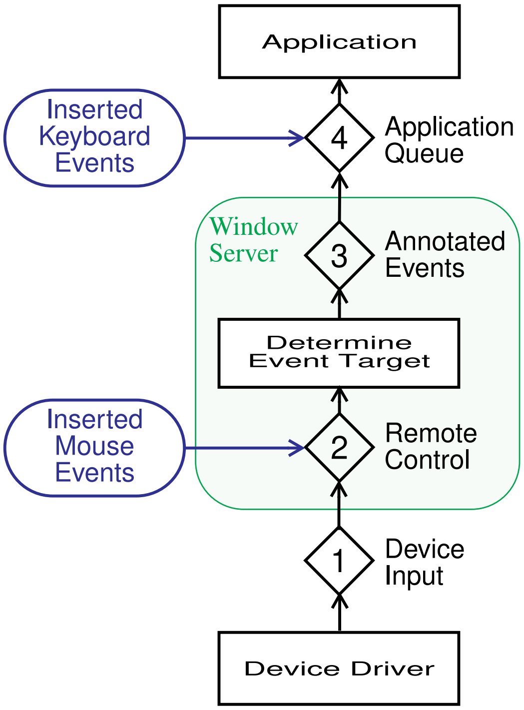

Figure 1: The Mac OS X event path from an input device to an application. Event taps are denoted by diamonds.

Sending mouse-click events to an arbitrary application, therefore, requires finding a point on the screen that is known to correspond to a window of the target application. For repeatability, we would like to generate events targeted at points in the relative coordinates of our target application's windows. However, Mac OS X effectively isolates applications from one another such that there is no way for one application to query the system about another application's windows or obtain a list of all windows open in the system (as we did on Windows). There is an accessibility API that supports some of the features required for fuzz testing, but it is currently unreliable and depends on explicit application support in some cases.

Lacking a way to consistently enumerate the windows of our target application, we simply generated random points in global screen coordinates and queried the system as to whether or not the points generated fell within the windows of our target application.

We developed an Objective-C tool, fuzz-aqua (named after Aqua, OS X's graphical user interface), for sending random streams of input events to an application. Our tool selects randomly from these basic event types, listed with the event primitives that implement them:

Keypress:	key down and up
Click:	mouse button down and up
DoubleClick:	click twice
Drag:	mouse button down, mouse move, mouse button up
ScrollWheel:	wheel movement (specific amount)

We also generate Keypress, Click, DoubleClick, and Drag events with arbitrary combinations of the modifier keys command, option, control, and shift.

While these combinations of events may seem simplistic – for example, a user typing quickly does not complete one keystroke at a time, but tends to overlap a few keystrokes in a row – we were not convinced that a more sophisticated input model would test applications more thoroughly, and in practice our simple model produced a considerable number of application failures.

We invoked fuzz-aqua using the command:

```
fuzz-aqua -d 0.05 -k "QMEsc" <PID> 100000
```

These options are described as follows:

- -d 0.05: Delay 50 ms between events
- -k "QMEsc": Do not send keyboard events with the keys 'Q', 'M', or Escape pressed in combination with the Cmd modifier key. This restriction prevents fuzz-aqua from minimizing or quitting the program it is testing, and prevents it from invoking some system-wide hot key commands, like logging out the current user.
- <PID>: The process ID of the applications to test.
- 100,000: The number of user-input events to send.

3 Experimental Method

We used the tools described in the previous section to test command-line and GUI-based applications on Mac OS X. We describe the applications that we tested, the test environment, and the tests that we performed. We then discuss how the data was collected and analyzed.

3.1 Platform and Applications

All of our tests were performed on Macintosh computers with version 10.4.3 of Mac OS X. We used two 867-MHz G4 PowerBooks and one Dual-2.5-GHz (single core) G5 PowerMac. Ideally we would have run every test on each machine to see whether any software failures could be correlated with different hardware configurations, but time constraints did not permit such a degree of thoroughness.

We tested 135 command-line applications with the fuzz input described in Section 2.1. We compared these applications to the 54 applications that were also tested on the Linux platform in our 1995 study. Note that some of the new applications that we tested are similar to applications tested in 1995, and we make our best attempt at providing a correspondence between the utilities tested in 1995 and in this current study. For example, groff is simply a front-end program for troff. Tables 1 and 3 list the applications tested and the test results for both our current study and the 1995 study. Table 1 lists the applications that crashed on one or both platforms and Table 3 lists the applications that did not crash.

We tested thirty graphical applications, using options described in Section 2.2. The applications that we tested are listed in Table 4. The results are dismal: twenty applications crashed, and two more hung (two of the crashing applications also hung during other test runs).

3.2 Failure Criteria

As in previous studies, our measure of reliability is simple and crude: the absence of a crash or hang. The command line tests run from scripts and check for the presence of a core file (crash) or non-responsiveness based on a time-out (hang). Of course, if the program completes without a crash or hang, but prints nonsensical results, we do not classify that as a failure. Other types of testing are better equipped for such failures.

For GUI applications, the technique is similar. We run the applications under fuzz-aqua, checking for a system-generated crash log or timing-out if the program hangs.

4 RESULTS

We first describe the basic quantitative success and failure results observed during our tests. The outcome of each test was classified in one of three categories: the application crashed completely, the application hung (stopped responding), or the application processed the input and was able to terminate via normal application mechanisms (i.e., exited under its own control). Since the categories are simple and few, we were able to categorize the success or failure of an application through simple inspection. We then provide analysis of the cause of failures for the applications for which we have source code.

4.1 Quantitative Test Results

Tables 1 and 3 summarize the command line testing results for Mac Table 1 lists the utilities that crashed in 1995 or 2006, and Table 3 lists the utilities that did not crash in either study. OS X and compares them to our previous 1995 results for Linux (the best of our study that year). Note that there were no command line tests run in our Windows study in 2000. The tables divide the utilities into two categories, those found only on our Mac OS X test machines and those found on both Mac OS X and on our previous Linux study. The Mac OS utilities are a strict superset of the ones tested in 1995, so there is no "Linux Only" category. We used the same amount and type of test data for testing both the Mac OS X utilities and the Linux utilities.

Each line in the table lists the utility's name on each platform and the test results. At the end of the table is a quantitative summary of the results. The Mac OS results are a summary of *all* utilities tested on that platform, totaling the results from both the "Only Mac OS" and "Both Mac OS and Linux" categories. The Linux results summarize the utilities tested in the "Both Mac OS and Linux" category. If an application failed on any of the runs in a particular category (column), the result is listed in Table 1. If the application neither crashed nor hung, it passed the tests.

The command line results for Mac OS X cover the largest number of utilities that we have ever tested, 135. Quantitatively, these results are as good as any of our previous studies; 7% of these utilities crashed and none hung in our tests.

Table 4 summarizes the results from our GUI application study. In this part of the study, we tested thirty application programs, more than we tested in the 1995 UNIX X-Windows study and similar to the number that we tested in the 2000 Microsoft Windows study. Of these thirty programs, a startling 73% (22) crashed or hung when presented with random valid keyboard and

Utility	Linux - 1995	Mac OS- 2006
Only Mac OS:		
expr		●
groff		●
zic		●
zsh		●
Both Mac OS and Linux:		
as		●
ctags	❍	
flex	●	
gdb	●	
indent	●	●
nroff		●
ditroff/troff		●
ul	●	●
vi(ex)/vim		●
Number crash/hang:	5	10
Number tested:	54	135
Percentage:	9%	7%

Table 1: Command Line Utility Results
● = crashed, ❍ = hung

Table 2:

Only Mac OS:
a2p, acid, aclocal, addftinfo, asa, automake, auval, bc, bridget, bsdmake, bspatch, bzip2, c2ph, c89, c99, calendar, checknr, column, dc, dd, diff3, ed, emacs, eqn, grn, gzip, h2ph, hexdump, jar, java, javac, javadoc, javap, jikes, ksh, lam, md5, merge, nano, native2ascii, neqn, od, osacompile, osascript, paste, pax, perl, php, pic, pl2pm, plutil, procmail, psed, pstopdf, python, rpcgen, ruby, script, sdiff, sdp, sliceprint, soelim, sqlite3, tab2space, tclsh8.4, test, texi2dvi, texi2html, unvis, units, uudecode, uuencode, vgrind, wall, wxPerl, xxd, yacc
Both Mac OS and Linux:
awk, bash, bison, compress, cat, col, colcrt, colrm, comm, cpp, csh, diff, expand, fmt, fold, ftp, gcc, grep, head, join, less, latex, look, m4, mail, make, nm, pr, refer, rev, sed, sort, split, strings, strip, sum, tail, tbl, tee, telnet, tex, tr, tsort, uniq, wc

Table 3: List of Utilities Tested that Did Not Crash or Hang

mouse events. This 73% is in comparison to 26% for X-Windows in our 1995 study, and 45% for Windows NT applications and 64% for Windows 2000 applications in our 2000 study. Because of the rapidly evolving collection of GUI applications over the last six years, and because of the changes in programs used for basic common tasks such as Web browsing and e-mail, we do not list side-by-side comparisons for individual applications.

4.2 Causes of Crashes

As in previous studies, we examined each program that crashed or hung to identify the cause of the failure. Source code was available to us for all the command-line programs we tested, but we had limited access to source code for the GUI-based programs. We had source for Aquamacs Emacs, Camino, Firefox, and Thunderbird, however we had build problems with Camino and Thunderbird, so were not able to debug them (this effort is ongoing). For each program failure where we had source, we categorized the cause using the same categories that we used in all the previous studies. Note that not all categories had failures in this study. These categories include:

- *Failure to check return values:* This mistake is an obvious and simple one – programmers often assume that a call can never fail or it is too much work or inconvenient to handle the case when it does fail. Unfortunately, these assumptions are often wrong, and the short term inconvenience can cause long term problems, i.e., short term gain for long term pain.
- *Pointer/arrays:* C and C++ encourage the use of pointers where other languages might use array subscripting, and bounds checking is considered to be in the “training wheels” category by serious programmers. Unfortunately, these features continue to be error prone, even in the hands of experienced programmers.
- *Signed characters:* The notion of an ASCII character as a 7-bit unsigned entity is certainly an over simplification, even if you only consider the traditional character set and do not consider more recent representations such as Unicode. The fact that the char type is a *signed* 8-bit integer does not help the situation.
- *Race conditions:* Assuming that operations will execute atomically, even in sequential programs, is dangerous. The classic example is receiving an interrupt (control-C, e.g.,) character while in the middle of processing the previous interrupt character.
- *Input functions:* Input functions without bounds checking, such as gets or the C++ >> operator (when used with variables of type `char*`), are inherently dangerous. Their reduction in use might be ascribed to the increased awareness of their role in security attacks such as stack smashing and buffer overruns.
- *Bad error handling:* Even when error checking is present, it is often ineffective. This problem is often due to the difficulty in generating test cases for all the obscure possible behaviors in a complex program.
- *Interaction effects:* This error is caused when a program intends to input a literal string, but the user embeds commands in the string and these commands are passed unintentionally to some interpreter (such as a format string or database query). This type of error has been a major source of recent security attacks.
- *Sub-processes:* Even if an applications’s code is iron-clad safe, it may be delegating control for some functionality to a sub-process. If this sub-process has any of the above problems, the application program also has these problems.

Vendor	Application	Result
Adobe	Acrobat Reader 7.0.5	●
adium.com	Adium X 0.87	●
Apple Computer	Calculator 10.4.3	
	Dictionary 10.4.3	❍
	Finder 10.4.3	●
	GarageBand 2.0.2	
	iCal 10.4.3	❍
	iChat 10.4.3	●
	iDVD 5.0.1	
	iMovie 5.0.2	
	iPhoto 5.0.4	
	iTunes 6.0.1	●
	Keynote 2.0.2	●
	Mail 10.4.3	●❍
	Pages 1.0.2	●
	Preview 10.4.3	
	Safari 10.4.3	
	Sherlock 10.4.3	●
	TextEdit 10.4.3	
	Xcode 2.2	●
aquamacs.org	Aquamacs Emacs 0.9.7	●
Microsoft Corporation	Excel 11.2.0	●
	Internet Explorer 5.2.3	●
	PowerPoint 11.2.0	●
	Word 11.2.0	●
Mozilla Foundation	Camino 0.8.4	●❍
	Firefox 1.5	●
	Thunderbird 1.5	●
Omni Group	OmniWeb	●
Opera Software	Opera 8.51.2182	●
	# tested:	30
	# crash/hang:	22
	%:	73%

Table 4: GUI Utility Results
● = *Crash,* ❍ = *Hang.*

The types of errors that we found and their relative frequency were roughly similar to our previous studies, with the exception of a reduction in input function errors.

In the remainder of this section, we describe our diagnosis of the failures that we found, grouping them by the above categories.

Function Return Codes

Not checking the return values of a called function would seem to be a true beginner mistake, but unfortunately is still present in modern code. In this code snippet from Aquamacs Emacs, the call to GetEventParameter (an event-handling system call) failed, returning an unchecked error code and an invalid window pointer in the output parameter wp (in file macterm.c):

```
#define mac_window_to_frame(wp) \
(((mac_output *) GetWRefCon (wp))->mFP)
...
GetEventParameter (event,
                   kEventParamWindowRef,
                   typeWindowRef, NULL,
                   sizeof (WindowRef),
                   NULL, &wp);
f = mac_window_to_frame (wp);
```

Given this invalid window-pointer as an argument, GetWRefCon returned a null pointer that was then dereferenced. The crash could have been avoided by checking GetEventParameter's return value and taking corrective measures.

Pointer/Array

Errors in the use of pointers and array subscripts still dominate the results of our tests. In all these cases, the programmer made implicit assumptions about the contents of the data being processed; these assumptions caused the programmer to use insufficient checks on their loop termination conditions or the values passed between functions in their program.

Reading more data than will fit into a statically allocated array is a classic example of this class of error. In the following segment from ul (in file ul.c), the array variable obuf is defined as a 512 element array, but its bounds are not checked. The program writes past the end of the array, corrupting maxcol. In the for loop the program eventually reads outside its memory range, causing a memory protection fault:

```
while ((c = getc(f)) != EOF) switch(c) {
   ...
   obuf[col].c_char = c;
   obuf[col].c_mode = mode;
   ...
   col++;
   if (col > maxcol)
      maxcol = col;
   continue;
}
...
for (i = 0; i < maxcol; i++) {
   if (obuf[i].c_mode != lastmode) {
   ...
```

Another way this problem crops up is in the use of sentinel characters. In these cases, the programmer assumes the input to their program will have a certain format. The program reaches a state where it is expecting a specific character before switching to another state, and may perform unsafe operations if the expected character never surfaces. In the following example from the time-zone compiler zic, the program has received an open quotation mark and is reading in more data expecting a matching close quotation mark before the end of the string (in file zic.c):

```
do {
  if ((*dp = *cp++) != '"')
    ++dp;
  else while ((*dp = *cp++) != '"')
    if (*dp != '\0')
      ++dp;
    else
      error(_("odd number of quotation marks"));
} while (*cp != '\0' &&
         *cp != '#' &&
         (!isascii(*cp) ||
           !isspace((unsigned char) *cp)));
```

The error in this case is especially pernicious, as it appears that the program will exit the loop when the end of the string is reached due to the call to error. In this case however, all that error does is print an error message and return to the loop. The program continues to overwrite memory until it finds a second quotation mark. Parenthetically, we note that naming a function simply as "_" should make the authors ashamed!

We also examined the open-source web browser Firefox after it crashed from random user input events. In Firefox 1.5, we encountered an unsafe dereference of a null pointer in file nsDocument.cpp, causing the application to crash:

```
nsIDocumentObserver *observer =
    NS_STATIC_CAST(nsIDocumentObserver *,
                   mObservers.ElementAt(i));
observer->ContentAppended(this,
    aContainer, aNewIndexInContainer);
```

The semantics of C++ make it difficult to see just what caused the problem here, so an explanation is in order. In this case, an element of mObservers was null, so when ContentAppended was called on this null object, the first access to a member variable of the object caused the program to crash.

While we do not know why mObservers contained a null element, this problem could have been avoided by checking the value returned from ElementAt.

A fourth example in this category is the utility expr, which crashed because the program ran off the end of its argument buffer:

```
int main(int argc, char **argv) {
   struct val     *vp;
   (void) setlocale(LC_ALL, "");
   av = argv + 1;
   if (!strcmp(*av, "--"))
   ...
```

When the program is called with no input parameters, av is set to a memory location past the end of the argv array. In this case, it points to the value of 0, and this null pointer is passed to the library function strcmp, which dereferences the null pointer and crashes.

Signed Characters

The conversion of numbers from one size to another can cause problems; this problem is compounded by using characters in both their symbolic and numeric forms. In C (and C++), the type “char” is a signed, 8-bit integer on most UNIX systems. The presence of a sign bit can be confusing and error prone (with the possibility of sign-extension) when doing arithmetic. The following example comes from as (file “expr.c”):

```
c_left = *input_line_pointer;
op_left = (operatorT)op_encoding[(int)c_left];
if(op_left == two_char_operator)
   op_left = two_char_op_encoding(c_left);
   while(op_left != O_illegal &&
          op_rank[op_left] > rank) {
      ...
```

The program reads an ASCII character from *input_line_pointer and stores it into the c_left variable. This variable is then sign-extended and used as an index into the op_encoding array. Since the sign-extended c_left is negative, op_left is assigned a garbage value. In this case, the garbage value is large, and when it is used as an index to the array op_rank in the while loop condition, the program crashes with a memory protection fault.

Bad Error Handling

A program’s error handler may detect bad input, but if the error handler is poorly written, it may itself cause a crash. The program troff detected an error in its input, but crashed while trying to print an error message due to an incorrectly escaped character in the error message string (node.cpp):

```
error("translation to \\% ignored in this
context");
```

This call eventually calls the function errprint, with the string transformed to “translation to \% ignored in this context” and stored in the variable format. The errprint function expects the ‘%’ symbol to be doubled, rather than escaped with a back slash (errarg.cpp):

```
while((c = *format++) != '\0') {
  if(c == '%') {
    c = *format++;
    switch(c) {
    case '%': . . .
    case '1': . . .
    case '2': . . .
    case '3': . . .
    default:
      assert(0);
    }
  }
}
```

The error handler finds a space character after the ‘%’, and the program asserts failure. Since this crash is due to a string hard-coded into the program, this error case was apparently never tested by the tool’s developers.

Sub-processes

An otherwise reliable program may crash if it invokes a less robust program as a sub-process. The crashes we encountered in groff and nroff occurred because those tools called troff, which crashed due to the problem described above when we ran them with the same test data that caused troff to crash

Other Causes

While we did not have source code for many of the Aqua applications that we crashed, we did examine the stack traces associated with the crashes. According to stack traces, nine of the twenty crashes occurred within system library functions. Three of these occurred within the function objc_msgSend_rtp (two of these involved calls to objects in the AppKit user inteface library), a low-level function integral to the Objective-C language runtime. Two occurred within methods of the khtml class in the WebCore HTML rendering library. One crash was in the function szone_calloc in the libSystem.B.dylib library. One occurred in function __bigcopy in the CoreFoundation framework. One occurred in the QuickDraw function SetGWorld, called from the dealloc method (similar to a C++ destructor) for the class NSQuickDrawView. The final crash occurred in function MDQueryDisableUpdates in the Spotlight file metadata library.

Without access to their source code, we could not determine whether these crashes were caused by invalid arguments to the functions or by other bugs within the libraries themselves. When the former is the case, crashes can be avoided by checking arguments before calling library functions; programmers should not assume that a library will gracefully handle erroneous input.

5 ANALYSIS AND COMMENTARY

As they say: *plus ça change, plus c’est la même chose* (the more things change, the more they stay the same). An optimistic view of software evolution would be that, as we learn more about the software development and engineering process, code should naturally get better. The pessimist (or perhaps the realist) would note that the commonly used programming languages and operating systems are not notably different from those that we used twenty years ago. In addition, software packages are providing more features and therefore are getting more complex. In such a view of the world, it is not surprising that the reliability of GUI-based applications is not improving, but instead seems to be getting worse.

Our evaluation of the stack traces from the Aqua application crashes showed that these problems seem to be wide spread and not caused by an isolated problem or vendor.

An additional force in this trend is the lack of demand for robust software. We, as consumers of software, continue to fixate on performance and features. Until there is a more global demand for more robust software, we cannot expect this situation to change. There is reason for hope though. The command line results, while not perfect, are good and have stabilized. And perhaps more notably, modern operating systems, while still far from being bug-free, crash much less often than those that we used twenty years ago. These systems have grown more complex and continue to use the same programming languages, but have managed to become more stable and robust.

The goal of this study was *not* to pick on Mac OS X. Instead, we used the availability of this (relatively) new system to revisit our basic evaluation techniques. Before condemning Apple for these results, a serious contemporary study of GUI-based applications should be done on UNIX and Windows (i.e., people in glass houses should not throw stones).

In 2000, we wrote:

> Will the results presented in this paper make a difference? Many of the bugs found in our 1990 UNIX study were still present in 1995. Our 1995 study found that applications based on open source had better reliability than those of the commercial vendors. Following that study, we noted a subsequent overall improvement in software reliability (by our measure). But, as long as vendors and, more importantly, purchasers value features over reliability, our hope for more reliable applications remains muted.

With each new release of fuzz testing results, we are often asked "do these bugs really matter". When we present our results to software developers and managers, we get a mixture of three basic responses:

1. *These bugs do not reflect realistic usage patterns, so it is not cost effective to spend time on fixing them.*

   In the early years of our studies, we did not have a good response to this criticism. However recent events have shown this view to be obsolete. The kinds of bugs that we find are true favorites for those who are developing security exploits. Even if we wish to ignore reliability as an end in itself, vulnerabilities in security have a clear cost. As Garfinkel and Spafford noted several years ago, reliability is the foundation of security [6]. And we quote from Microsoft, "An insecure driver is inherently unreliable, and an unreliable driver is inherently insecure." [13]

2. *This crash data is easy to obtain and free, and the crashes might occur in actual use, so the bugs should be fixed.*

   This is the view that we hope to hear. It is rarer than we would like. Note that by "easy to obtain and free", we mean that the test programs, results, and fixes are available on our web site.

3. *These are bugs!* Bugs! *My code is malformed and I must fix it!*

   This view is one that resonates with those who see software development as a true craft, and not just a job. A woodworker, artist, or stonemason would be loath to produce a work with a known flaw, and would cringe at the thought of not fixing such a flaw if one was pointed out to them. You either get this view or you do not. These are the folks that we would like to have working for or with us.

Our study would be more complete with more complete access to source code. We would be happy to diagnose failures for any package in Table 4 that the vendor supplies to us.

6 Related Work

Random testing has been used for many years. In the past, it has been looked upon as primitive by the testing community. In his book on software testing [12], Myers says that randomly generated input test cases are "at best, an inefficient and ad hoc approach to testing". While the type of testing that we use may be *ad hoc*, we do seem to be able to find bugs in real programs. Our view is that random testing is one tool (and an easy one to use) in a larger software testing toolkit. The body of related work on random testing is huge, and we present only a part of it here (and apologize to those authors whose papers we slighted).

An early paper on random testing was published by Duran and Ntafos [4]. In that study, test inputs were chosen at random from a predefined set of test cases. The authors found that random testing fared well when compared to the standard partition testing practice. They were able to track down subtle bugs easily that would otherwise be hard to discover using traditional techniques. They found random testing to be a cost effective testing strategy for many programs, and identified random testing as a mechanism by which to obtain reliability estimates. Our technique is both more primitive and easier to use than the type of random testing used by Duran and Ntafos; we cannot use programmer knowledge to direct the tests, but do not require the construction of test cases.

Two papers have been published by Ghosh et al. on random black-box testing of applications running on Windows NT [7,8]. These studies are extensions of our earlier 1990 and 1995 Fuzz studies [14,15]. In the NT studies, the authors tested several standard command-line utilities. The Windows NT utilities fared much better than their UNIX counterparts, scoring less than 1% failure rate. This study was interesting, but had two significant limitations. First, they only tested a few applications (attrib, chkdsk, comp, expand, fc, find, help, label, and replace) and second, they did not test the most commonly used Windows applications that based on graphic interfaces. We tested Windows GUI-based (Win32) applications in 2000 [5].

Random testing has also been used to test the UNIX system call interface. The "crashme" utility [3] effectively exercises this interface and is actively used in Linux kernel developments.

In recent years, many projects have developed systems for testing software using structured random input data, in contrast to the unstructured or minimally structured input data that our fuzz utilities produce. McKeeman describes a hierarchy of structure for random test inputs in the context of compiler testing [11]. He shows how different aspects of a C compiler are tested when the input data has differing degrees of conformance to a valid C program. McKeeman also describes the technique of differential testing, in which different programs that perform the same function are run using the same input and their outputs compared with differences in output hopefully signaling a bug in one of the programs. Sirer and Bershad [16] used a context-free grammar to generate structured random input in the form of programs in Java bytecode for use in testing Java virtual machines.

Random input and black-box testing has also been used extensively in network protocol testing, and has been effective in finding security vulnerabilities in protocol implementations. The SPIKE [1] project produces a software package that supports automated black-box testing of network protocols. Marquis et al. provide a language for describing network protocols to generate well-formed input data for those protocols that can then be mutated in order to test the protocol implementations [10]. Xiao et al. describe a system for injecting invalid data into network protocols to test for robustness failures and security vulnerabilities [18].

Random testing is not just for software. Wood et al. found random testing to be effective while designing multiprocessor cache controller hardware [17]. They developed a utility to generate random memory accesses to a simulated cache controller and found over half of the functional bugs in their design during simulation. After fabricating prototype hardware, they were able to continue testing with the same software in order to verify their design.

In 1983, before the Macintosh was released, Apple developed a tool similar to fuzz-aqua called "The Monkey" [9]. The developers of application software for the new computer were having trouble reproducing certain bugs that only occurred in situations where little memory was available. They developed a utility that took advantage of a demonstration feature of the operating system to produce random user input events and send them to the current application. By using this utility, they were able to find and fix many of these difficult to reproduce bugs. This utility saw less use as computers started to ship with more memory, but this sort of tool is no less useful today.

7 Conclusion

The goal of our fuzz testing work has been threefold. First, it started as an effort to explain a phenomenon that we observed on that dark and stormy Wisconsin night. However, this work has taken on a life of its own. Our second goal has been to provide a simple test of techniques and tools to add to the reliability of software. And third, we hope to provide some concrete measure, albeit crude, of how well we are doing in achieving software reliability.

We know that our measure of reliability is a primitive and simple one. This is both a strength and weakness. The weakness is that we exploit no knowledge of the semantics of a program nor do we explicitly test how well a program matches its specification. While the criterion is crude, it offers a mechanism that is easy to apply to almost any application, and, we believe that any cause of a crash or hang should not be ignored in any program.

Fuzz testing, in its many guises, has become part of the argot of the testing, security, and intelligence communities. This broad acceptance is perhaps our best accomplishment.

On-line Resources

The previous papers, test results, and source and binary code for the fuzz tools for this and previous studies are available from our Web page at:

`http://www.cs.wisc.edu/~bart/fuzz`.

Acknowledgments

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176 and ONR contract N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] D. Aitel, "The Advantages of Block-Based Protocol Analysis for Security Testing", Immunity Inc., February 2002. http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html

[2] Apple Computer, May 2006, http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/chapter_2_section_6.html.

[3] G.J. Carrette, "CRASHME: Random Input Testing", http://people.delphi.com/gjc/crashme.html, 1996.

[4] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering* **SE-10**, 4, July 1984, pp. 438-444.

[5] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", *4th USENIX Windows Systems Symposium*, Seattle, August 2000. Appears (in German translation) as "Empirische Studie zur Stabilität von NT-Anwendungen", *iX*, September 2000.

[6] S. Garfinkel and G. Spafford, **Practical UNIX & Internet Security**, O'Reilly & Associates, 1996.

[7] A. Ghosh, V. Shah and M. Schmid, "Testing the Robustness of Windows NT Software", *1998 International Symposium on Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, November 1998.

[8] A. Ghosh, V. Shah and M. Schmid, "An Approach for Analyzing the Robustness of Windows NT Software", 2*1st National Information Systems Security Conference*, Crystal City, VA, October 1998.

[9] A. Hertzfeld, **Revolution in the Valley**, O'Reilly Media, Inc., Sebastopol, CA, 2004, pp. 184-185.

[10] S. Marquis, T. Dean, S. Knight, "SCL: a Language for Security Testing of Network Applications", *2005 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, October 2005.

[11] W. McKeeman, "Differential Testing for Software", *Digital Technical Journal*, Digital Equipment Corporation **10**, 1, December 1998.

[12] G. Myers, **The Art of Software Testing**, Wiley Publishing, New York, 1979.

[13] Microsoft Corporation, "Security and Reliability Strategies", http://www.microsoft.com/whdc/driver/security/, 2006.

[14] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", University of Wisconsin-Madison, 1995. Appears (in German translation) as "Empirische Studie zur Zuverlasskeit von UNIX-Utilities: Nichts dazu Gerlernt", *iX*, September 1995. ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisted.pdf.

[15] B. P. Miller, L. Fredriksen, B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM* ***33***, 12, December 1990, pp. 32-44. Also appears in German translation as "Fatale Fehlerträchtigkeit: Eine Empirische Studie zur Zuverlassigkeit von UNIX-Utilities", *iX* (March 1991). ftp://grilled.cs.wisc.edu/technical_papers/fuzz.pdf.

[16] E. Sirer and B. Bershad, "Using Production Grammars in Software Testing", *Symposium on Domain-Specific Languages*, Austin, TX, October 1999.

[17] D. Wood, G. Gibson, and R. Katz, "Verifying a Multiprocessor Cache Controller Using Random Case Generation", *Computer Science Tech report UCB/CSD-89-490*, University of California, Berkeley, 1989.

[18] S. Xiao, L. Deng, S. Li and X. Wang, "Integrated TCP/IP Protocol Software Testing for Vulnerability Detection", *2003 International Conference on Computer Networks and Mobile Computing*, Shanghai, October 2003.