Taint-based Directed Whitebox Fuzzing *

Vijay Ganesh† and Tim Leek‡ and Martin Rinard†
†MIT Computer Science and Artificial Intelligence Lab, ‡MIT Lincoln Laboratory
vganesh@csail.mit.edu, tleek@ll.mit.edu, rinard@csail.mit.edu

Abstract

We present a new automated white box fuzzing technique and a tool, BuzzFuzz, that implements this technique. Unlike standard fuzzing techniques, which randomly change parts of the input file with little or no information about the underlying syntactic structure of the file, BuzzFuzz uses dynamic taint tracing to automatically locate regions of original seed input files that influence values used at key program attack points (points where the program may contain an error). BuzzFuzz then automatically generates new fuzzed test input files by fuzzing these identified regions of the original seed input files. Because these new test files typically preserve the underlying syntactic structure of the original seed input files, they tend to make it past the initial input parsing components to exercise code deep within the semantic core of the computation.

We have used BuzzFuzz to automatically find errors in two open-source applications: Swfdec (an Adobe Flash player) and MuPDF (a PDF viewer). Our results indicate that our new directed fuzzing technique can effectively expose errors located deep within large programs. Because the directed fuzzing technique uses taint to automatically discover and exploit information about the input file format, it is especially appropriate for testing programs that have complex, highly structured input file formats.

1 Introduction

Fuzz testing [17] is a form of automatic, black-box testing which uses a *fuzzer* to randomly generate or mutate sample inputs. This technique has been shown to be surprisingly effective in exposing errors in software systems [17, 23, 16]. It is especially useful for testing input parsing components — the randomly generated inputs often exercise overlooked corner cases in the initial parsing and error checking code.

But fuzz testing (or simply fuzzing) has been less effective at generating syntactically legal inputs that expose deeper semantic errors in programs [23, 8] — for many programs virtually all of the randomly generated inputs fail to satisfy the basic syntactic constraints that characterize well-formed inputs, and hence fail to make it past the initial parsing phase to exercise the remaining code.

This paper presents a new testing approach, *directed whitebox fuzz testing*, and a new tool, *BuzzFuzz*, that implements this testing approach. Instead of generating random inputs that primarily exercise the initial input parsing components, directed fuzz testing is designed to produce well-formed test inputs that exercise code deep within the core semantic processing components of the program under test. As such, it complements random fuzzing and significantly extends the reach of automated testing techniques. Directed fuzz testing is based on the following techniques:

- **Taint Tracing:** The program under test executes on one or more valid sample inputs. The execution is instrumented to record taint information. Specifically, the instrumented execution records, for each value that the program computes, the input bytes that influence that value.
- **Attack Point Selection:** Specific points in the program under test are identified as potential *attack points*, i.e., locations that may exhibit an error if the program is presented with an appropriate error-revealing input. By default, our implemented *BuzzFuzz* system selects library and system calls as attack points. *BuzzFuzz* can also be configured, under user control, to select any arbitrary set of program points as attack points.
- **Directed Fuzzing:** For each attack point and each sample input, *BuzzFuzz* computes the set of input bytes that affect the values at that attack

*This research was supported in part by National Science Foundation grants CCR-0325283, CNS-0509415, and CCF-0811397, and the Department of Defense under the Air Force Cooperative Agreement FA8750-06-2-0189. The Lincoln Laboratory portion of this work was sponsored by the Department of Defense under the Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

point. For library and system calls, for example, the values at the attack point are the parameters passed to the library or system call. *BuzzFuzz* then generates new inputs as follows: Each new input is identical to one of the sample inputs, except that the input bytes that affect the values at one or more attack points have been altered. By default, our implemented *BuzzFuzz* system sets these bytes to extremal values, e.g., large, small, or zero integer values. *BuzzFuzz* can also be configured to use different policies such as generating random values for the corresponding input bytes.

- **Directed Testing:** Finally, *BuzzFuzz* runs the program under test on the newly generated inputs to see if the inputs expose any errors.

This approach has several benefits:

- **Preservation of Syntactic Structure:** Directed fuzzing tends to target input bytes that can be changed without violating the legal syntactic structure of the original seed inputs. The automatically generated fuzzed inputs therefore tend to make it past the initial parsing components to exercise code within the semantic core of the computation.
- **Targeted Values:** The altered input bytes are designed to target values that are directly relevant to specific potential vulnerabilities. The generated test suite therefore tends to have a high concentration of inputs that can expose errors that may exist at these potential vulnerabilities.
- **Coordinated Changes:** Finally, directed fuzzing can identify and alter multiple disjoint regions of the input space that must change together in a coordinated way to expose the presence of an error.

One of the keys to making this approach work is choosing a set of attack points that tend to have latent errors. We have found that library calls can comprise a particularly productive set of attack points. The program and library are typically developed by different people operating with different perspectives, information, and assumptions. This cognitive gap between the program and library developers can easily enable a range of errors — the program may use the library in ways that the library developer did not anticipate, corner cases can become lost or forgotten at the interface between the program and the library, or the program developer may simply be unaware of some library preconditions. Moreover, the large potential execution space at library calls can be difficult to adequately explore with standard testing practices, making library calls particularly susceptible to the kind of latent errors that our directed fuzzing technique is designed to expose.

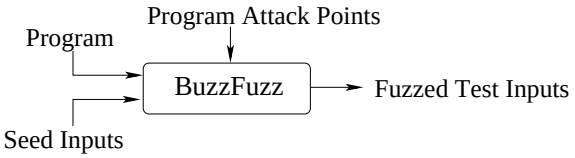

Figure 1. *BuzzFuzz* **Inputs and Outputs. The inputs to** *BuzzFuzz* **are: source of a C program, seed inputs, and a list of program attack points. The output of** *BuzzFuzz* **is a set of fuzzed test inputs for the program under test.**

Experimental Results: We have used *BuzzFuzz* to test several programs that process highly structured binary data (examples of such data include video, images, document, and sound files). We have found that directed fuzzing is able to successfully generate inputs that satisfy complex input consistency constraints and make it past the initial input processing components to expose errors at attack points within the semantic core. More specifically, our results show that *BuzzFuzz* is able to preserve the syntactic structure present in the original inputs, generate new inputs that satisfy the key properties that characterize legal inputs, and successfully target vulnerabilities deep within the semantic core of the program under test. In the context of programs for processing highly structured inputs like movies or documents, *BuzzFuzz* nicely complements random fuzzing, since random fuzzing usually uncovers errors in shallower parts of the program such as input validation code.

All of the experimental results presented in this paper are available at `http://people.csail.mit.edu/vganesh/buzzfuzz.html`. This web site includes the source code for our benchmark applications, the automatically generated test files that expose errors in these applications, and the seed test input files that *BuzzFuzz* started with to generate the test files.

Inputs and Outputs: Figure 1 illustrates the program inputs and outputs of the *BuzzFuzz* system. The user provides *BuzzFuzz* with a program and set of seed input files for that program. *BuzzFuzz* also accepts a specification of attack points. *BuzzFuzz* comes preconfigured with a default attack point specification; The user can also provide additional attack point specifications. Given these inputs, *BuzzFuzz* instruments the source to trace taint information, then runs the instrumented source on the provided inputs to find out which

input file locations influence attack point values. It then uses this information to produce new fuzzed test input files.

Contributions: This paper makes the following contributions:

- **Technique:** It presents a new automatic technique for using taint information to fuzz input files. This technique uses the taint information to identify promising locations in the input file to fuzz, while preserving the syntactic structure of the original input file.
- **Results:** It presents experimental results that characterize the effectiveness of our automatic technique on two sizable open-source applications. These results show that our technique is effective in generating inputs that make it past the initial input parsing components to automatically expose subtle errors within the semantic core of programs that have complex input file formats.

2 Example

We next present an example that illustrates the basic concepts and operation of *BuzzFuzz*. The example presents an error that *BuzzFuzz* exposed in Swfdec, an open source C program that decodes and renders Adobe Shockwave Flash movie and animation formats [21, 1]. Figure 2 presents two procedures: the `jpeg_decoder` procedure from the Swfdec Version 0.5.5 source code and the `clipconv8x8_u8_s16_c` procedure that Swfdec invokes from the libOIL library [2]. libOIL is an optimized inner loop library; it contains a variety of simple functions that have been optimized to take advantage of the extended instruction set features in the microprocessor. Swfdec uses various libOIL functions to improve the performance of its image processing operations.

In the example code in Figure 2, the Swfdec `jpeg_decoder` procedure reads in JPEG images embedded inside Flash movie files, decodes the images, and populates the datastructure `dec` that holds the resulting decoded images. As part of this computation, it invokes the libOIL `clipconv8x8_u8_s16_c` procedure to convert 8 by 8 arrays of signed 16 bit integers into corresponding 8 by 8 arrays of unsigned 8 bit integers. Each 8 by 8 destination array is embedded within a larger array that holds the converted image blocks. Because this larger array stores the decoded image, its size depends on the size of the image that Swfdec is currently processing.

Before the call to `jpeg_decoder`, Swfdec stored the width and height of the image in `dec->width` and `dec->height`. Swfdec originally read these values in from the (syntactically valid) Flash input file. During the setup for the image decode, these values

```
1  //Application Code
2  //Swfdec JPEG Decoder
3  jpeg_decoder(JpegDecoder* dec){
4   ...
5   dec->width_blocks =
6   (dec->width + 8*max_h_sample - 1)/
7   (8*max_h_sample);
8   dec->height_blocks =
9   (dec->height + 8*max_v_sample - 1)/
10  (8*max_v_sample);
11  int rowstride;
12  int image_size;
13  ...
14  rowstride =
15  dec->width_blocks * 8*max_h_sample/
16  dec->compsi.h_subsample;
17  image_size = rowstride *
18  (dec->height_blocks * 8*max_v_sample/
19  dec->compsi.v_subsample);
20
21  dec->ci.image=malloc(image_size);
22  ...
23  //LibOIL API function call
24  clipconv8x8_u8_s16_c(dec->ci.image...);
25  ...
26 }
27 //End of Application Code
28
29//Library Code
30 clipconv8x8_u8_s16_c(ptr...){
31   ...
32    for (i = 0; i < 8; i++) {
33      for (j = 0; j < 8; j++) {
34
35        x = BLOCK8x8_S16 (src,sstr,i,j);
36        if (x < 0) x = 0;
37        if (x > 255) x = 255;
38
39        //CRASH POINT!!!!!
40        (*((uint8_t *)((void *)ptr +
41        stride*row) + column)) = x;
42      }
43    }
44  }
45 //End of Library Code
```

Figure 2. Swfdec JPEG decoder (jpeg.c line 523), and libOIL Code. Highlighted code indicates dynamic taint transfer from width and height of JPEG image to `clipconv8x8_u8_s16_c`, the libOIL library API. An out of bounds memory access causes a crash at CRASH POINT.

were copied into the `dec` data structure that Swfdec uses to store the decoded image.

Taint Tracing: Figure 2 shows in boldface the flow of dynamic taint from `dec->width` and `dec->height` (line numbers 5 and 8, respectively, in the source code shown in Figure 2) to the `ptr` parameter of `clipconv8x8_u8_s16_c` (line number 30).

As this flow illustrates, the taint flows from `dec->height` and `dec->width` through the computation of the `image_size` value to the `ptr` parameter of `clipconv8x8_u8_s16_c`. As part of its dynamic taint trace computation, *BuzzFuzz* records that input bytes 0x1843 through 0x1846 in the original input Flash file (these bytes contain the image height and width that Swfdec read and copied into `dec->height` and `dec->width` for that particular input) influence the parameter values at the call to `clipconv8x8_u8_s16_c`. *BuzzFuzz* also records the fact that Swfdec interprets input bytes 0x1843 through 0x1846 as integer values.

Fuzzed Test Input Generation: Because the call to `clipconv8x8_u8_s16_c` is an attack point, *BuzzFuzz* fuzzes the corresponding input bytes 0x1843 through 0x1846 when it constructs its new test input. Specifically, *BuzzFuzz* uses the fact that Swfdec interprets the input bytes as integers to set the height and width to the extremal value 0xffff in the new test input files. Bytes that do not affect values at attack points are left unchanged.

Execution on Test Input: When Swfdec attempts to process the new fuzzed test input, the computation of `image_size` overflows, causing the call to `malloc` to allocate a destination image array that is too small to hold the decoded image. The resulting out of bounds array writes in `clipconv8x8_u8_s16_c` cause Swfdec to fail with a SIGSEGV violation.

Discussion: This example illustrates several features of the *BuzzFuzz* system. First, the use of taint tracing enables *BuzzFuzz* to automatically find input file bytes that it can change without destroying the syntactic structure of the original input file. In our example, the taint tracer identifies which bytes in the input file represent the integer height and width. This information enables *BuzzFuzz* to apply targeted changes that preserve the underlying syntactic structure of the input Flash file. The result is that the new fuzzed input Flash file makes it past the initial input parsing components to exercise components (like the `jpeg_decode` procedure) in the core of the computation.

Second, this use of taint information enables *BuzzFuzz* to operate in a completely automatic, push-button fashion. In effect, *BuzzFuzz* observes how the program itself manipulates the input file bytes to discover how to change the input without invalidating the underlying synactic structure of the original input file. There is no need for the *BuzzFuzz* tester to specify or even understand the input file format. *BuzzFuzz* is therefore especially appealing for applications (such as Swfdec) with very complex input file formats. On the one hand, traditional random fuzzing techniques have great difficulty producing syntactically valid input files for such programs, and are therefore primarily useful for testing the input file parsing components of such programs. On the other hand, more structured techniques that assume the presence of some specification of the input file format require the tester to understand and specify the input file format before effective testing can begin [23, 12, 8]. By automatically discovering enough information about the input file format to enable effective directed fuzz testing, *BuzzFuzz* makes it possible to automatically obtain test cases that exercise the core of the computation (rather than the input parsing) without requiring the tester to understand or specify the input file format.

Finally, the example illustrates how library calls can provide relevant attack points. Many modern programs make extensive use of libraries, with the core data structures (as in Swfdec) passed as parameters to libraries. These library parameter values are often determined by specific combinations of input bytes (such as the image height and width in our example) that the program uses to compute the shape and form of its internal data structures. In many cases (as in Swfdec), related distinct values must be changed together in a coordinated way to expose the underlying error in the code that processes the data structures. Choosing library calls as attack points is one productive way to identify these kinds of related combinations of input file bytes.

3 Technique

The *BuzzFuzz* system, as illustrated in Figure 1, requires three inputs:

- The source code of a program P written in the C language,
- A list of attack points in the form of a list of function names (our production *BuzzFuzz* system comes with several lists preconfigured to include specific library and system calls), and
- One or more seed inputs I for the program P.

Given these inputs, *BuzzFuzz* automatically and without any human intervention produces new test inputs T that are derived from I by appropriately fuzzing input bytes that affect values at attack points according to the types of these attack point values.

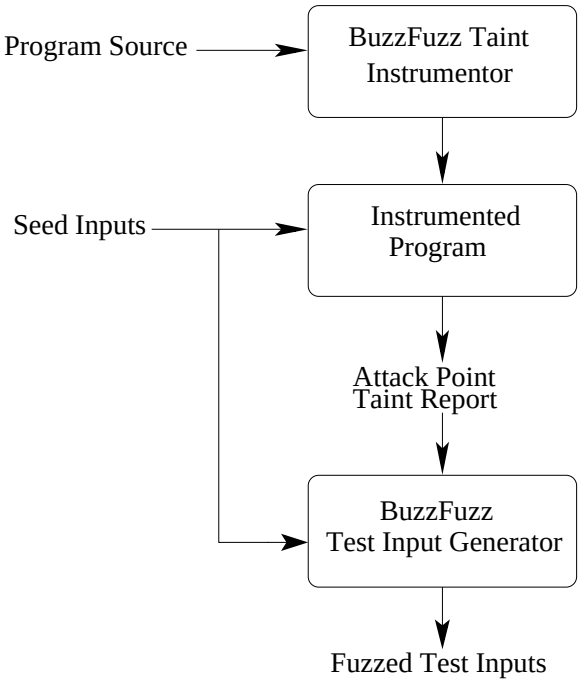

Figure 3. Internals of *BuzzFuzz*

Figure 3 graphically illustrates the operation of *BuzzFuzz*. The *BuzzFuzz* taint instrumentator takes as input the source of the program under test. It produces as output the instrumented program, which runs on the provided seed test inputs. The result of each instrumented execution is a *BuzzFuzz* attack point taint report which indicates how specific locations in the input file influence values at attack points. Finally, the *BuzzFuzz* test input generator uses the taint reports and the seed inputs to produce the fuzzed test inputs. We next describe the individual *BuzzFuzz* components in more detail.

3.1 Dynamic Taint Tracing

The *BuzzFuzz* taint tracing implementation processes the source code of the program under test to appropriately insert calls to the *BuzzFuzz* taint trace library. This library maintains a mapping that records, for each computed value, the input bytes that affect that value. This mapping is implemented as a hash table that maps addresses to sets of byte locations in the input file. Whenever the program uses a set of existing values to compute a new value, the inserted taint tracing calls use the hash table to retrieve the input byte locations for the existing values used to compute the new value, take the union of these sets of input byte locations, then record this set of input byte locations in the hash table indexed under the address where the new value is stored.

For example, consider the case where the program under test uses an assignment statement to copy a value from an input buffer into a local variable. The inserted taint tracing call causes the *BuzzFuzz* dynamic taint tracing system to retrieve the set of input byte locations for the accessed input buffer address, then store that set of input byte locations in the hash table indexed under the address of the assigned local variable. Statements that combine multiple values to compute a new value (for example, a statement that adds several values) take the union of the input byte locations for all of the values used to compute the new value.

Type Information: In addition to the file location (or input byte location) information, the taint tracing system also records the type of the new value. *BuzzFuzz* uses this information during its directed fuzzing step to choose appropriate extremal values for the fuzzed input file bytes.

Procedure Calls: To trace taint across procedure calls, *BuzzFuzz* maintains a stack that records information about the taint information for procedure parameters. At each procedure call point, the inserted calls to the taint trace library push taint information for the parameters onto the stack. The taint trace calls inside the invoked procedure then use this stack to retrieve taint information for statements that use the parameters.

Library Procedures: In certain cases *BuzzFuzz* does not have access to the source code for procedures in libraries. *BuzzFuzz* is therefore unable to insert the taint tracing calls into these procedures. *BuzzFuzz* instead maintains an internal database of information about how taint flows across a small set of important library procedure calls. For example, *BuzzFuzz*'s database specifies that taint flows from the parameter of `malloc` to its return value. *BuzzFuzz* uses this information to insert appropriate calls to the taint library at such procedure call points.

Attack Points: At each attack point the inserted *BuzzFuzz* instrumentation records the taint information for all of the values at the attack point each time the attack point executes.

Report: When the execution terminates, *BuzzFuzz* produces a report that contains a list of input bytes (and the corresponding types) that influence values at attack points. Together, the entries in this list identify every region of the input that influences a value that appears at an attack point. We present several lines from a Swfdec report below:

```
((int) 0x1862 0x1863 )
((int) 0x1862 0x1863 0x1864 0x1865 )
```

Limitations: As is standard for most taint tracing systems of which we are aware, *BuzzFuzz* does not trace indirect tainting relationships that occur at conditionals or array accesses. So if a particular set of input bytes is used to determine whether to execute the true or false branch of a conditional, the locations of those bytes may not appear as influencing the values computed along the executed side of the conditional. Similarly, for array assignments taint flows only from the array index to the assigned array element and not to the remaining unmodified array elements.

Performance: The inserted instrumentation imposes a performance overhead. Specifically, our results indicate that the taint tracing calls cause the instrumented version of the program to execute between 20 and 30 times slower than the original non-instrumented version. As the results in Section 3.3 indicate, this performance overhead does not significantly impair the ability of our system to generate test inputs that expose errors in our benchmark applications.

3.2 Generating Fuzzed Inputs

The first step in generating fuzzed inputs is to acquire a corpus of seed inputs. Given such a set, *BuzzFuzz* runs the instrumented version of the program on each input in turn. The result is a report identifying which bytes in the input influence values that appear at attack points.

The next step is to use the report to fuzz the corresponding seed input to obtain a new test input. Each report entry identifies a sequence of input bytes that *BuzzFuzz* can change along with the type of the value stored in that set of bytes. Many possible fuzzing strategies are possible — for example, the new value for the bytes may be chosen at random or from a predefined set of values. The current implementation of *BuzzFuzz* chooses extremal values such as very large or small values. The specific bit-level representation of the value is chosen based on the type recorded in the report.

The current implementation fuzzes every identified sequence of input bytes to produce, for each seed input, a single fuzzed input. If entries conflict (i.e., specify overlapping but not equal byte ranges or conflicting types), *BuzzFuzz* arbitrarily applies changes from only one of the entries. It is, of course, possible to use more involved strategies that fuzz some but not all of the identified input bytes. It is also possible to generate multiple fuzzed test input files from a single seed file (for example, by fuzzing different subsets of input byte sequences, by applying different fuzzing strategies to the same input byte sequence, or by combining these techniques).

3.3 Testing

The last step is to run the uninstrumented version of the program on the new fuzzed test inputs and observe any failures. These failures can then be reported to developers.

4 Results

We experimentally evaluate *BuzzFuzz* on two open-source C applications, namely, Swfdec [21] version 0.5.5, an Adobe Flash player, and MuPDF [4] version 0.1, a PDF viewer.

4.1 Methodology

We first obtained an initial corpus of seed input files for each application. We would like to thank Pedram Amini for providing us with a corpus of Adobe Flash files and Adam Kiezun of MIT for providing us with a corpus of Adobe PDF files. We verified that each application processes all of the files in its corpus correctly without failures.

We next used the corpus as a basis for directed fuzz testing. Specifically, we provided *BuzzFuzz* with 1) an instrumented version of the application program under test (this version uses dynamic taint tracing to compute the input bytes that affect each value at each attack point), 2) the corpus of seed input files for that application, 3) the uninstrumented production version of the application, and 4) a specification of the attack points. *BuzzFuzz* then iterated through the files in the corpus of seed input files, performing the following steps for each file:

- **Instrumented Execution:** The instrumented version of the application executes on the seed input file. This execution produces a report that identifies, for each attack point, the input bytes that influence the values that occur at that attack point and the types of these values.
- **Directed Fuzzing:** *BuzzFuzz* processes both the report generated in the step above and the seed input file to produce a new fuzzed test file. This test file differs from the seed input file in only those bytes that affect values at attack points. Those bytes are set to extremal values, with the specific extremal value chosen as a function of the type of the corresponding attack point value.
- **Testing:** The uninstrumented version of the application executes on the automatically generated fuzzed test file. All failures are logged and information (such as the test input file and stack back trace at the point of failure) recorded.

We performed two sets of experiments. The first set identified all calls to the X11 library as attack points.

Application	Lines of Code	Number of Fuzzed tests	Total Errors	Distinct Errors	Mean Stack Depth	Errors per hour	Distinct Errors per hour
Swfdec	70,000	2152	128	5	28	11.33	0.41
MuPDF	40,000	553	1	1	7	.25	0.25

Table 1. *BuzzFuzz* **Results**

Application	Crash Type	File:#	Library	Stack Depth
Swfdec	BADALLOC	XCreatePixMap	X11 Library	23
Swfdec	BADALLOC	XCreatePixMap	X11 Library	23
Swfdec	SIGABRT	`cairo_pen.c`:325	CAIRO Library	43
Swfdec	SIGSEGV	`convert8x8_c.c`:130	LibOIL Library	40
Swfdec	SIGSEGV	`swfdec_sprite_movie.c`:377	Swfdec Application	11
MuPDF	SIGSEGV	atoi	libc Library	7

Table 2. Information on Specific Errors Exposed by *BuzzFuzz*

The second set identified all calls to the Cairo and LibOIL libraries as attack points. Using multiple focused sets of attack points (as opposed to a single large set that includes all attack points) has the dual advantages of 1) fuzzing more precisely targeted portions of the input file (thereby promoting the preservation of the legal syntactic structure of the seed input files), and 2) minimizing the possibility that errors exposed at one set of attack points will mask errors exposed at other attack points. For Swfdec, for example, a combined set of attack points that includes calls to the X11, Cairo, and LibOIL libraries produces a set of fuzzed inputs that exposes errors only at calls to the X11 library. For this set of inputs, the application always fails at an X11 call before it can reach additional errors exposed at calls to the Cairo and LibOIL libraries.

4.2 Experimental Results

Table 1 presents our results for both Swfdec and MuPDF. The first column presents the number of lines of code in each application (excluding libraries), the second presents the total number of fuzzed test inputs presented to each application, the third presents the number of executions that failed, the fourth presents the number of distinct errors responsible for these failures (in some cases a single error was responsible for failures in multiple different runs), and the fifth presents the mean depth of the stack trace when the failure occurred. The last two columns present the number of failed executions per hour and the number of distinct errors found per hour.

We note that the error discovery rate per hour of testing reflects the overall effectiveness of our technique. In general, the mean stack depth trace numbers reflect the fact (discussed further below) that the errors occur deep within the core of application rather than in shallower, more peripheral components.

Table 2 presents additional detail for each error. There is one row in the table for each error that *BuzzFuzz* exposed. The first column presents the application with the error; the second column identifies the manifestation of the error as a specific crash type. The next column identifies the location of the manifestation of the error, either as a filename plus line number or (if that information is not available) as a function name. The next column identifies this location as either in a specific named library (such as LibOIL, CAIRO, X11 [2], or libc) or within the source code of the application itself. The next column identifies the depth of the stack at the manifestation point.

The information in this table indicates that *BuzzFuzz* (configured with attack points at library and system calls) can effectively expose errors related to problematic interactions across module boundaries. Specifically, five of the six errors manifest themselves within library rather than application code.

The BADALLOC errors, for example, are caused by the application passing very large width (x-axis) and height (y-axis) window size parameters to the X11 graphics library. The memory management system is unable to satisfy the library's ensuing attempt to allocate a data structure to hold the (very large) window's contents, and the program fails because of a BADALLOC error.

The CAIRO SIGABRT error similarly involves a problematic interaction between the application and the library. This error occurs when a field specifying the number of strokes in a given image representation is inconsistent with the actual number of strokes. This inconsistency does not effect the application until it is deep within the CAIRO library actually using the number of strokes as it attempts to render the image. Also, as discussed above in Section 1, the SIGSEGV error in the libOIL library occurs because of an over-

Application	Lines of Code	Number of Fuzzed tests	Total Errors	Distinct Errors	Mean Stack Depth	Errors per hour	Distinct Errors/hour
Swfdec	70,000	9936	198	3	19	16.5	0.25
MuPDF	40,000	555	4	1	7	2	0.5

Table 3. Random Fuzzing Results

Application	Crash Type	File:#	Source	Stack Depth
Swfdec	SIGSEGV	`swfdec_sprite_movie.c`:377	Swfdec Application	11
Swfdec	SIGABRT	`swfdec_as_types.c`:500	Swfdec Application	22
Swfdec	BADALLOC	`swfdec_as_context.c`:967	Swfdec Application	18
MuPDF	SIGSEGV	atoi	libc Library	7

Table 4. Information on Specific Errors Exposed by Random Fuzzing

flow when computing the size of a memory block used to hold converted JPEG images.

These errors reflect the difficulty of making libraries with complex interfaces robust in the face of inconsistent or simply unusual or unanticipated parameter settings. Many of the errors involve corrupted metadata that is passed (potentially after translation) from the input file through the application into the library, which either fails to check the validity of the metadata or simply does not have enough information to do so. The application itself, on the other hand, is often acting primarily as a conduit for the information and is in no position to perform the detailed checks required to detect parameter combinations that may cause the library to fail. In effect, the division of the computational task into an application plus a library has left an information gap that provides room for errors to occur.

4.3 Comparison with Random Fuzzing

To compare directed and random fuzzing, we implemented a random fuzzer that reads in an arbitrary input file, then randomly fuzzes bytes within the file to generate new fuzzed test files. Like the directed fuzzing technique that *BuzzFuzz* uses, the purely random fuzzer provides a fully automatic, push-button testing environment. Unlike directed fuzzing, however, the random fuzzer does not attempt to exploit any information about how the application processes input file bytes. It therefore implements a less targeted testing strategy.

The random fuzzer can be configured to fuzz a chosen percentage of the input files. For our set of benchmark applications, randomly fuzzing bytes in the file headers almost always produces an invalid file that is immediately rejected by the inital input parsing code. Moreover, fuzzing over 20 percent of the remaining input file bytes almost invariably produces an immediately rejected invalid file. Fuzzing less than 5 percent of the remaining input bytes, on the other hand, produces minimally perturbed files that fail to expose any errors at all. We therefore chose to randomly fuzz 10 percent of the non-header bytes in the input files.

Table 3 presents our results for random fuzzing for both Swfdec and MuPDF. The first column presents the number of lines of code in each application (excluding libraries), the second presents the total number of fuzzed test inputs presented to each application. Unlike *BuzzFuzz*, the random fuzzer does not need to execute an instrumented version of the application on the seed input files. It can therefore generate new fuzzed input files much more quickly than *BuzzFuzz*, which in turn enables it to test the applications on more input files (we ran both testers for 12 hours). The third column presents the number of executions that failed, the fourth presents the number of distinct errors responsible for these failures (in some cases a single error was responsible for failures in multiple different runs), and the fifth presents the mean depth of the stack trace when the failure occurred. The last two columns present the number of failed executions per hour and the number of distinct errors found per hour.

This random fuzzing strategy exposed three distinct errors in Swfdec and one error in MuPUDF. Table 4 presents additional information for each of the errors. Some of the errors overlap with the errors in Table 2 that *BuzzFuzz* exposed. Specifically, the first error in Table 4 and the third error in Table 2 are the same error. Also, the MuPDF errors are the same for both fuzzing techniques.

The remaining errors, however, illustrate the different and complementary strengths of the two techniques. The remaining two errors that the random fuzzer exposed occur in the initial parsing of the Flash action script language embedded within the Flash input file. In fact, both errors manifest themselves as assertion failures inside the Swfdec source code. They therefore trigger anticipated and handled fatal error conditions (and are therefore arguably not errors at all). On the other hand, the remaining four Swfdec errors from Table 2 that *BuzzFuzz* exposed all occur

within the core of the computation after the input has passed the initial input parsing phase. These differences reflect the effectiveness of directed fuzz testing in 1) identifying parts of the input file that, when appropriately fuzzed, expose errors within the core of the program while 2) preserving the underlying syntactic structure of input files with complex formatting constraints to enable the automatically generated test files to make it past the initial parsing and input validation to exercise the error-containing code within the core.

Note that because directed fuzz testing tends to preserve the synactic structure of the seed input files, it is not designed to expose errors in the initial input parsing code. And in fact, the results show that simple random fuzzing is more effective at appropriately exercising the parsing code to expose such errors.

5 Related Work

Fuzzing: Fuzzing refers to a class of techniques for randomly generating or mutating seed inputs to get new test inputs. These techniques have been shown to be surprisingly effective in uncovering errors [17, 23], and are used heavily by security researchers. Fuzzing is relatively cheap and easy to apply. However, it suffers from several drawbacks: many random inputs may lead to the same bug, and the probability of producing valid inputs may be low, especially for deeply structured formats like movies. Furthermore, the probability of exposing certain classes of incorrect behavior, which require many conditions to be simultaneously true in deep program paths, can be vanishingly small.

Our directed fuzzing technique provides similar ease of use benefits. But because it exploits information about how the program accesses the input file bytes to preserve the important syntactic structure present in the initial seed inputs, it can effectively target deep errors in the core of the computation. The tradeoff is that it is not designed to expose errors in the initial input parsing code.

Grammar-based Black Box Fuzzing: Black box fuzzing tools use a grammar to characterize syntactically legal inputs [23, 3, 12]. The insight behind this approach is that a fuzzer that understands the input file format can preserve the syntactic validity of the initial seed inputs (or even inputs generated from scratch) and therefore produce test inputs that make it past the initial input parsing code to exercise code in the core of the computation.

Our directed fuzzing technique is also designed to preserve the syntactic validity of the seed inputs to produce fuzzed inputs that expose deep errors in the core of the computation. But because directed fuzzing exploits the availability of taint information to effectively identify and change appropriate parts of the seed input file, it achieves this effect without requiring the tester to understand or obtain a specification of the input file format. In general, the cost of obtaining this specification can be substantial, especially for files (such as movie, image, and document files) with complex file structure. For example, see [23], Chapter 21 for an analysis of the cost of developing a grammar-based fuzzer for the Adobe Flash file format.

Another advantage is that the tainting information enables directed fuzzing to target multiple regions in the input file that must be changed together in a coordinated way to expose the error. Because the computation tends to combine the values of such regions when it builds its internal data structures, this grouping information is readily available in the taint reports that *BuzzFuzz* produces.

Concolic Testing: Generally speaking, a concolic tester executes the subject program both concretely and symbolically on a seed input until it reaches some interesting program expression [10, 22, 7, 18]. The concrete execution serves the purpose of choosing a program path cheaply, while the symbolic part of the execution is converted into a constraint, called a *path constraint*. The interesting program expression under consideration could be a program assertion, a conditional, or a dangerous expression like division. The path constraint is conjoined with a query about this program expression (e.g., can the assertion be violated, can negation of the conditional lead to a viable alternate path, or can the denominator of the division go to zero), and fed to a constraint solver for solution. The solution, in terms of variables representing the input, is a test case that can exercise the program path and the program expression in interesting ways, potentially exposing an error. Concolic testing has been shown to be effective, has the advantage of being systematic and is usually completely automatic.

However, concolic testing faces several challenges [5, 9]. First is the exponential explosion in the number of shallow paths in the early part of the code (usually parser code) that are systematically explored by concolic testers. In many cases, and especially for programs that process highly structured input files, the constraint solver gets bogged down exploring the many shallow execution paths that lead to parser errors. One way to ameliorate this problem is to augment the concolic tester with a grammar that specifies the legal input file format [11]. By using the grammar to guide the constraint satisfaction, it is possible to avoid exploring many error cases in the input parsing code. As for grammer-based black box fuzzing, a potential drawback is the need to obtain a grammar that characterizes the legal inputs.

The second issue is that, once the concolic tester makes it past the initial input parsing stages, the resulting deeper program paths may produce very large constraints with complex conditions that current state of the art constraint solvers are unable to handle. This is especially problematic for deep program paths that contain hard to invert functions like hash functions or encryption.

Both concolic testing and *BuzzFuzz* use the program source to generate some kind of symbolic information for each program variable in a particular concrete program run. However, the key difference is in the kind of symbolic information that each technique maintains. In the case of concolic testing, the symbolic information is essentially a logical expression for each variable that semantically captures all possible values these variables can take for the particular program path chosen by the concrete execution. In contrast, *BuzzFuzz* simply maintains the set of input bytes that influence the program variables in the particular program path chosen, in particular for the variables involved in the attack points. In other words, *BuzzFuzz* uses a simpler and more tractable algebra of symbolic expressions, i.e., sets of input bytes per program variable, as opposed to concolic testers, which maintain logical expressions per program variable.

This distinction in the kind of symbolic information maintained by each technique is a key differentiator. For programs whose inputs are highly structured, concolic execution of deep program paths may result in heavyweight constraints that are difficult or even impossible to solve. *BuzzFuzz*, on the other hand, works with much lighter weight symbolic information and is therefore capable in practice of exploring much deeper program paths and exposing errors that occur deeper in the computation. The trade-off, of course, is that *BuzzFuzz* cannot systematically enumerate inputs that exhaust the program execution space — the probability of exposing an error depends entirely on the interaction between the fuzzing algorithm and the characteristics of the program.

Dynamic Monitoring Using Taint Tracing: Dynamic taint tracing has also been used to find potential security vulnerabilities and monitor deployed programs for the presence of potential security attacks. The idea is to use the taint information to detect situations in which the program uses input data in a potentially dangerous way (such as the target of a branch instruction) [13, 19, 20].

It is also possible to use taint tracing to improve test coverage. Comet dynamically tracks taint from regions of the input to conditionals in the subject program [14]. It then uses heuristics to determine new values for the input regions that may result in the program taking a new path, thus potentially increasing coverage.

Protocol Induction: The goal of automatic protocol induction or protocol reverse engineering is to automatically *infer* the file format or the grammar for valid inputs of a given program [6, 24, 15]. Like our technique, these techniques use taint tracing to obtain information about the structure of the input file. Unlike our technique, which aims to expose errors by modifying targeted parts of the input file, the goal of protocol induction is simply to obtain an understanding of the input format.

6 Conclusions

In comparison with other testing approaches, random testing offers clear advantages in automation, ease of use, and its ability to generate inputs that step outside the developer's expected input space. Our new directed fuzz testing technique, as implemented in our *BuzzFuzz* tool, complements existing random testing techniques to enable, for the first time, fully automatic generation of test inputs that exercise code deep within the semantic core of programs with complex input file formats.

A key insight behind the success of our technique is the use of taint tracing to obtain information about how input bytes affect values that appear at attack points. This taint information helps *BuzzFuzz* generate new, syntactically valid fuzzed input files, thereby enabling *BuzzFuzz* to successfully target specific attack points deep within the computation. It also enables *BuzzFuzz* to identify multiple regions of the input file that must be modified in a coordinated way to expose errors. Because the taint information is available automatically without human intervention (or even any need for the tester to understand any aspect of the input file format), it enables the fully automatic generation of precisely targeted test inputs. This property makes it especially appropriate for testing programs with complex input file formats.

Testing is currently the most effective and widely used technique for enhancing program robustness and reliability. By opening up a new region of the testing space for automatic exploitation, our directed fuzz testing technique promises to help developers find and eliminate deep subtle errors more quickly, efficiently, and with less effort.

Acknowledgements

We are grateful to Pedram Amini and Adam Kiezun for providing us with a large number of Flash and PDF files, respectivley, for testing purposes. We would also like to thank Adam Kiezun and David Molnar for their useful feedback on early drafts of the paper.

References

[1] Adobe macromedia shockwave flash file format. http://en.wikipedia.org/wiki/Adobe_Flash.

[2] Gnome and freedesktop enviroments. http://en.wikipedia.org/wiki/Freedesktop.org.

[3] Wikipedia entry on fuzzing. http://en.wikipedia.org/wiki/Fuzz_testing.

[4] T. Andersson. Mupdf: A pdf viewer. http://ccxvii.net/fitz/.

[5] P. Boonstoppel, C. Cadar, and D. R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *TACAS*, pages 351–366, 2008.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, New York, NY, USA, 2007. ACM.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, October-November 2006.

[8] J. DeMott. The evolving art of fuzzing. http://www.vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf, 2006.

[9] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed Systems Security Symposium*, 2008.

[12] R. Kaksonen. A functional method for assessing protocol implementation security. Technical Report 448, VTT Electronics, 2001.

[13] E. Larson and T. Austin. High coverage detection of input-related security faculties. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[14] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann. Coverage maximization using dynamic taint tracing. Technical Report TR-1112, MIT Lincoln Laboratory, 2007.

[15] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, Atlanta, GA, USA, November 2008.

[16] B. Miller. Fuzzing website. http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html, 2008.

[17] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[18] D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report UCB/EECS-2007-23, University of California, Berkeley, CA, Feb 2007.

[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on comnodity software. In *NDSS*, 2005.

[20] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP Security*, 2005.

[21] B. Otte and D. Schleef. Swfdec: A flash animation player. http://swfdec.freedesktop.org/wiki/.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, 2005.

[23] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 1 edition, July 2007.

[24] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS08*, 2008.