Software Engineering Institute

Probability-Based Parameter Selection for Black-Box Fuzz Testing

Allen D. Householder
Jonathan M. Foote

August 2012

Carnegie Mellon

Table of Contents

List of Figures

List of Tables

Abstract

Dynamic, randomized-input functional testing, or *black-box fuzz testing*, is an effective technique for finding security vulnerabilities in software applications. Parameters for an invocation of black-box fuzz testing generally include known-good input to use as a basis for randomization (i.e., a seed file) and a specification of how much of the seed file to randomize (i.e., the range).This report describes an algorithm that applies basic statistical theory to the parameter selection problem and automates selection of seed files and ranges. This algorithm was implemented in an open-source, file-interface testing tool and was used to find and mitigate vulnerabilities in several software applications. This report generalizes the parameter selection problem, explains the algorithm, and analyzes empirical data collected from the implementation. Results of using the algorithm show a marked improvement in the efficiency of discovering unique application errors over basic parameter selection techniques.

1 Introduction

Dynamic randomized-input functional testing, also known as *black-box fuzz testing* or *fuzzing*, has been widely used to find security vulnerabilities in software applications since the early 1990s [1, 2, 3]. Since then, fuzz testing evolved to encompass a multitude of software interfaces and a variety of testing methodologies [4, 5, 6].

Because of their basic nature, black-box fuzzing techniques and tools are relatively simple to implement and use. However, black-box fuzzing has known disadvantages when compared to more sophisticated techniques—notably inferior code path coverage and reliance on the selection of a good set of seed input (e.g., seed files) [4, 5]. Despite advances in fuzzing tools and methodologies, many security vulnerabilities in modern software applications continue to be discovered using these relatively unsophisticated techniques [7, 8, 9, 10, 11].

Studies that compare fuzzing methodologies generally recommend using a mix of methodologies to maximize the efficacy of vulnerability discovery [5, 7, 12, 13, 14]. Our experience with black-box fuzz testing showed that the difference between an effective fuzzing effort (i.e., one that finds vulnerabilities) and an ineffective one (i.e., one that does not) often lies in the selection of parameters passed to the fuzzing tool.

In this report, we present research focused on automating parameter selection for a sustained black-box fuzz testing effort. The algorithm presented here was implemented in the open-source CERT® Basic Fuzzing Framework (BFF) product [15] and was used to discover several previously unknown security vulnerabilities [8, 9, 10, 11]. Although our work was implemented to test application file interfaces running on Unix operating systems, it is notionally applicable to other fuzzing tools, operating systems, and interface types.

This report covers the following topics:
- We describe a workflow for black-box fuzz testing that maximizes the number of unique application crashes found during a sequence of test iterations.
- We identify two parameters for an iteration of file fuzz testing and generalize the problem of parameter selection.
- We present an algorithm to be used for selecting fuzzing parameters that maximize the number of unique application errors.
- We discuss the results of executing an implementation of the algorithm on several applications and compare them to the results of executing the same number of tests run without the algorithm.

® CERT is a registered trademark owned by Carnegie Mellon University.

2   Background: Black-Box Fuzz Testing with the CERT BFF

The CERT BFF is a system used for testing the security of applications on Unix-based (e.g., Linux, Mac OS X) operating systems. The CERT BFF uses Sam Hocevar's zzuf tool [16] to perform mutation-based, black-box fuzz testing on application file interfaces. The zzuf tool in turn executes the application under test. We refer to successive invocations of zzuf testing a single application as a *fuzzing campaign*. The CERT BFF allows a security auditor to perform a fuzzing campaign by automating invocations of the zzuf tool (see Figure 1).

The zzuf testing tool is open source software, so detailed user documentation is publicly available [16, 17]. Each invocation of the zzuf tool repeatedly executes the application under test with mutated test cases until a crash is detected or until an optional maximum number of application executions exit without a crash. In this report, we refer to each execution of the application under test (via zzuf) as an *iteration of fuzz testing*. We refer to successive iterations executed under a single invocation of zzuf as an *iteration interval*. In the CERT BFF, the maximum interval size passed to zzuf is configurable, but the default is 500 iterations.

In addition to the application under test, the maximum iteration interval size, and other options, an invocation of zzuf must also specify a seed for randomization (*randomization seed*), a path to a file to use as a basis for mutation (*seed file*), and a proportion of the file to randomize (*range*). The zzuf tool randomly mutates the seed file by changing a number of bits roughly equal to the specified range multiplied by the size of the seed file. The number and position of the bits that are changed is randomized based on the randomization seed. Randomization in zzuf is implemented so that repeated invocations using the same randomization seed, range, and seed file will produce identical test cases. In this report we refer to the range and seed file collectively as *fuzzing parameters*.

A fuzz campaign in the CERT BFF is a loop built around successive invocations of the zzuf tool (see Figure 1). First, the CERT BFF chooses zzuf invocation parameters for the next iteration interval by examining the running set of crashing test cases (if any exist). The algorithm used to choose the zzuf invocation parameters is the focus of subsequent sections of this report.
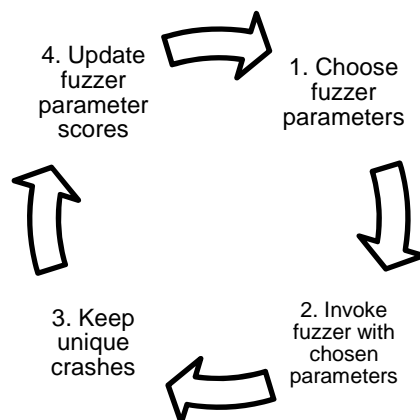
Figure 1:   Main Loop for a Fuzz Campaign in the CERT BFF

Second, the CERT BFF invokes zzuf with the chosen parameters. If a crash is detected during that iteration interval, the application is launched using a debugger to generate a hash of the application back trace at the point of failure. The logic used to generate this hash is extended from the fuzzy stack hash method employed in a study researching dynamic test generation to find bugs in Linux programs [5].

The hash determines if the detected crash represents a unique application error for the fuzz campaign. Finally, if the new hash is unique, it is added to a running set, the parameter scores are updated, and additional analysis is performed on the new unique crash. Besides hashes, the running set also includes metadata such as the crashing test case, the seed file from which it was derived, and the fuzzing parameters used to find it.

Initial selection of seed files [4], application and system configuration [18], triage of application errors [19], and application error analysis are important aspects of a fuzzing campaign. However, these aspects are not directly relevant to the fuzzing parameter selection algorithm outlined in this report, so we do not discuss them in detail.

3 Maximizing the Number of Unique Crashes

In this section, we describe two aspects of the parameter selection problem and show how the application of a generalized solution can improve the results of a fuzzing campaign.

Recent fuzzing frameworks and research use file format grammars, static analysis, white-box approaches, and other techniques to direct fuzzing based on code execution paths [5, 7, 20, 21, 22]. This new approach is simpler because we begin a campaign with very little knowledge about the seed files, format details, or code coverage. Instead, we apply basic probability theory to adjust parameter selection as the campaign progresses to maximize the number of unique crashes found during a fuzzing campaign. This section first describes how the CERT BFF attempts to reach this goal for seed files, then ranges, and then generalizes the algorithm to apply to both parameters.

3.1 Selecting Seed Files

To find as many unique crashes as we can, we begin by assuming no knowledge of the seed files. We simply empirically measure their *crash density*, which we define as follows: for the ith seed file, the crash density d_i is the empirically measured number of unique crashes found while fuzzing the file (u_i) divided by the number of trials attempted (t_i).

$$\textit{equation 1} \qquad\qquad\qquad d_i = \frac{u_i}{t_i}$$

The goal is to measure the crash density of a seed file, then allocate the fuzzing campaign's resources to focus the bulk of the effort on the seed files that are most productive (i.e., have a higher crash density). Thus we expect the selection probability across the seed file to be set to vary over time as the campaign finds (or does not find) new unique crashes within the set. We do not abandon seed files entirely, though, because it is possible that they could produce unique crashes.

The desired solution has the following four criteria:

1. A seed file with observed crash density d should be chosen twice as often as a seed file with an observed crash density of $d/2$.
2. If the crash density of two files is equal, either one should be chosen with equal probability.
3. Seed files with no observed crashes should not be abandoned because they could still produce unique crashes.
4. Given two files, both with 0 observed crashes, the one with fewer trials should be preferred over the one with more trials because we know more about the one with more trials.

3.2 Modeling the Process

We begin by modeling the fuzzing process as a sequence of Bernoulli trials. Each trial is independent and can have one of two possible outcomes: either the program crashes or it does not. We leverage this simple model to equate the crash density of individual seed files with the probability of finding a crash in the next iteration.

In the CERT BFF, an interval may contain a few hundred to a few thousand iterations at a time to mitigate setup and teardown costs (in terms of both CPU and person hours) associated with invoking the fuzzing tool. For each invocation of the fuzzing tool, we choose the next seed file with a probability proportional to its observed crash density relative to the other seed files' crash densities. Therefore, to make this selection, we first calculate the sum D of all the crash densities:

$$\text{equation 2} \qquad\qquad D = \sum_{i=1}^{n} d_i$$

Then we calculate a probability distribution on the set of seed files S such that each seed file s_i is chosen with probability pi given by

$$\text{equation 3} \qquad\qquad p_i = \frac{d_i}{D}$$

For the start-up case in which both u_i and t_i are zero, which would leave d_i undefined, we simply assign the value $d_i=1$. This approach is essentially what is called "fitness-proportionate selection," as described by John Holland in his book *Adaptation in Natural and Artificial Systems* [23].

At this point, we satisfied the first two criteria for the desired solution listed in Section 3.1. For two files with $u1=6$ and $u2=3$ and an equal number of trials $t1=t2=100$, we can see that $p1=2p2$. Also, if $d1=0.02$ and $d2=0.02$, then $p1=p2$. However, we still must deal with the case where some number of trials was completed but no crashes were observed as in the third criterion. As it stands, if $u_i=0$, then $d_i=0$ and thus $p_i=0$.

Because we are describing a series of Bernoulli trials, we expect the results to conform to a binomial distribution. However, since the number of trials is much larger than the expected number of successes (a single success out of thousands of trials is typical), we can apply the Poisson approximation of the binomial distribution to allow for easier calculation of confidence intervals.

The Poisson distribution is described by the parameter λ: the number of successes divided by the number of trials. Hence the crash density d_i can be substituted for the parameter λ in a Poisson distribution. Doing so allows us to take advantage of confidence intervals on the Poisson distribution to address the third criterion.

We calculate the one-sided 95% upper confidence bound on the Poisson distribution UCB (d_i, 95%) to be used in lieu of the crash density d_i for each file. Substituting into equation 2, it becomes

$$\text{equation 4} \qquad\qquad D = \sum_{i=1}^{n} UCB(d_i, 95\%)$$

and therefore a modified p_i is also used such that

$$\text{equation 5} \qquad\qquad p_i = \frac{UCB(d_i, 95\%)}{D}$$

Because the 95% upper confidence bound is always greater than zero, we no longer have a problem with files that have no observed crashes. They will be chosen with a non-zero probability. Furthermore, since the width of the confidence interval narrows, given more trials and the same λ (or d_i), the final criterion is addressed because two files, both having no observed crashes, will result in preferring the one with fewer trials due to its larger confidence interval value.

The following procedure summarizes the algorithm thus far:

1. Given a set of seed files, assign each file an initial crash density of 1.0 (i.e., assume they will definitely crash on the next try).
2. Calculate p_i for each file (p_i for all files will be equal at this point).
3. Choose a file according to the distribution given in step 2.
4. Fuzz the file for an interval of iterations (e.g., a few hundred trials).
5. Calculate the upper 95% confidence interval on the observed crash density for that file (i.e., update p_i).
6. Repeat the steps, starting with step 2.

3.3 Specifying Ranges

Different file formats have different structural requirements. Some are merely a header followed by data. Others have higher structural requirements (e.g., XML, PDF). Fuzzing tools typically have either a set value or require the analyst to choose a parameter that specifies how much input mutation should be performed. As previously described, zzuf provides a parameter called *range* that allows the user to specify the proportion of bits that will be flipped in the test cases it generates.

Ranges are expressed as a decimal value between 0.0 and 1.0 that indicates the proportion of the file that will be altered (e.g., a range of 0.06 would mean that 6% of the bits will be flipped when the file is read). The range parameter allows a lower and upper limit to be specified, so a user can instruct zzuf to fuzz between 10% and 50% of the bits in the file using the range value 0.1–0.5.

Using various fuzzing tools, we learned that if too many bits are flipped, the file format may no longer be recognized as valid. The program may reject the file as invalid before processing data that would reveal defects. At the same time, fuzzing too few bits in the file inadequately searches the space of possible inputs. We also observed that while there can be a significant difference between fuzzing 0.5% of the bits in a file versus 1%, there is little difference between fuzzing 90% and 90.5% of the file. Thus, we divide the range 0.0–1.0 into exponentially scaled ranges starting with a range that effectively fuzzes one or two bits up through a range that fuzzes between 60–100% of the file.

Once we calculate the various ranges we plan to use, we still must decide how to allocate the fuzzing campaign across those ranges. A naïve implementation would be to simply select a range with equal chance of choosing any given range. Instead, we applied the same method used for seed file selection to measure each range's crash density. This approach permits us to automatically adjust the fuzzing campaign's range parameters on each seed file based on what the campaign actually found. Different files may have different optimal ranges for any number of reasons, so it is best to adjust the range parameters for individual files rather than for a set of files.

The algorithm is nearly identical to that presented for seed files:

1. Given a set of ranges for a seed file, assign each file an initial crash density of 1.0 (i.e., assume they will crash on the next try).

2. Calculate p_i for each range (they will all be equal at this point).

3. Choose a range according to the distribution given in step 2.

4. Fuzz the file for an interval of iterations (i.e., a few hundred trials).

5. Calculate the upper 95% confidence interval on the observed crash density for that range (i.e., update p_i).

6. Repeat the steps, starting with step 2.

3.4 Generalizing the Algorithm

We successfully applied this algorithm to the tuning of a fuzz campaign's parameters of seed file selection and range selection. As a result, we formulated a generalized model, which we refer to as a *scorable set*. The scorable set maintains a set of items, each of which keeps track of its number of successes and trials, which it, in turn, uses to calculate its 95% upper confidence bound to be used in probability calculations.

We expect to use this model for other parameter selection problems where we lack a more robust model. For example, in a fuzzing campaign where multiple mutation strategies (e.g., bitwise, bytewise) are used, a scorable set with the mutation strategies as the individual items could be used to automatically measure and discover the most productive mutation strategy. A simplified example of the scorable set model implemented in Python can be found in *Appendix B: Scorable Set Example Source Code (Python)*.

4 Methodology

In this section, we describe the methodology for testing the approach described in Section 3.

The new algorithm was initially developed using the CERT BFF to fuzz ImageMagick's *convert* program version 5.2.0. This older version of *convert* is known to be highly susceptible to fuzzing (i.e., it crashes frequently). Each iteration of fuzz testing completes quickly, so it provides a quick turnaround for testing new algorithms. The initial results from *convert* were promising.

To more conclusively determine whether the algorithm for parameter selection could reveal more unique application errors than completely random parameter selection, we designed an experiment to test two applications. We executed scenarios with the algorithm enabled and scenarios without the algorithm enabled for both applications. If the scenarios using the new algorithm revealed more unique crashes than the ones without it, we would conclude that the algorithm improved the detection of unique application errors over the default (purely random) scenario.

To confirm the results of the experiment, we set up two distinct fuzzing campaigns using the CERT BFF: one using FFmpeg version SVN-r0.5.5-4:0.5.5-1 and the other using Oracle's Outside In 8.3.5 library [13, 24]. We chose these two products because they offer a large attack surface stemming from their support for many different types of files as input.

FFmpeg is a tool used for converting, recording, and streaming audio and video. Outside In provides the ability to read and transform various unstructured file formats. For FFmpeg, we used a collection of various video files found on the internet as seed files. Similarly, the Outside In tests used a variety of document and other file types that Outside In normally handles. Table 1 describes the seed files we used in more detail.

Table 1: Application Test Summary

	FFmpeg	Outside In
Version	SVN-r0.5.5-4:0.5.5-1	8.3.5
Number of Seed Files	104	76
Seed File Types	AVI, F4V, FXM, MNG, MPG, MP4, OGV, MOV, SWF, ASF, WMV	123, AI, ASF, EML, EXE, DLL, CDR, DCX, DPT, DWG, EMF, GDF, ICO, ICS, JB2, JP2, JPX, MHT, MGS, ODG, ODT, OFT, ONEPKG, PCX, PDF, PICT, PPM, PPT, PR4, QPW, MOV, RTF, RM, SDA, SDC, SDD, SDW, SHW, SVG, SXC, SXD, SXI, SXW, TXT, TGA, TIF, DAT, VCF, AVI, VSD, WB1, WB2, WB3, WBMP, WMV, WK1, WK2, WK3, WK4, WMF, WPG, XPS, XSN
Seed File Sizes	1KB-960KB (median 12KB)	75B-3.9MB (median 48KB)

For each application, we created four scenarios:

1. equiprobable selection of both seed files and ranges (denoted S0 R0 in Tables 2 and 3)

2. equiprobable seed file selection with scorable-set-based range selection (S0 R1)

3. scorable-set-based seed file selection with equiprobable range selection (S1 R0)

4. scorable-set-based seed file and range selection (S1 R1)

Finally, because the fuzzing process is inherently stochastic, for each of the four scenarios we created 5 virtual machine clones to allow us to analyze the variability of the results.[1] Each virtual machine clone was then configured to use a different initial randomization seed value to be passed to zzuf via the CERT BFF. In total, there were 20 virtual machines per application. The virtual machines were allowed to run for 18 days for FFmpeg and 6 days for Outside In. Examples of the CERT BFF configuration for both campaigns are listed in *Appendix A: CERT BFF Configuration*.

---

[1]   We used the Debian-based "DebianFuzz" virtual machine distributed with the CERT BFF version 2.5. This virtual machine is designed to provide a host environment specifically tuned for fuzzing. By default, it expects a single processor and 500MB of RAM, so it is sufficiently lightweight to allow many clones to exist on a single physical system.

5   Results and Discussion

This section describes the results of the controlled test conducted using the approach described in Section 4.

Results for both applications indicate that after an initial learning period where the algorithm explores the parameter space, the scenario in which both seed file and range selection use the scorable set algorithm outperforms the other scenarios.

At the start of a campaign, the seed files and ranges are all equally likely to be selected because they all have equal crash density values of 1. After the first seed file and range are selected and an iteration interval is completed, their crash densities reflect the measured value (as already described), which is typically much smaller than 1.

Because of the selection algorithm, a specific seed file and range will likely not be selected again until all other ranges and seed files receive some attention. For a fuzzing campaign with dozens of seed files and a few dozen ranges for each seed file, this start-up phase can take a few thousand iterations to find the first crash, assuming it finds any at all. The findings from the experiment indicate that all four scenarios behave similarly until after a few crashes are found.

For FFmpeg, after 500,000 iterations the scenario with both selection features enabled yielded an average of 121 unique crashes versus an average of 106 unique crashes for the equiprobable scenario (14% improvement). At 1 million iterations, the score was 210 to 143 for both features versus equiprobable, respectively (47% improvement). At 5 million tests, the score was 616 to 333, respectively (85% improvement).

The mixed scenarios with either range selection or seed file selection enabled fell in the middle at just over 400 unique crashes after 5 million tests. Table 2 summarizes the results for the FFmpeg campaign.

Table 2:   Experiment Results Summary for FFmpeg

FFmpeg	Unique Crashes Found (Average of 5 Runs Each)			
N Trials	S0 R0	S0 R1	S1 R0	S1 R1
500	0	0	0	0
1K	0	0	0	0
5K	3	3	2	4
10K	6	7	6	8
50K	23	23	23	23
100K	38	33	39	41
500K	106	95	112	121
1M	143	151	172	210
5M	333	421	413	616

The results are shown more visually in Figure 2. Notice the clear advantage of using the S1R1 scenario.
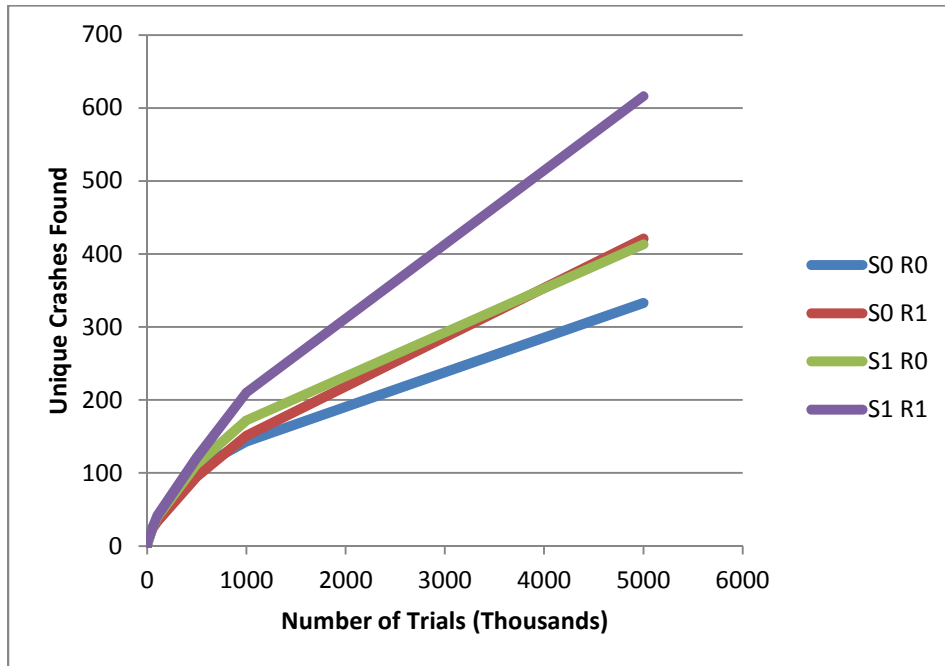
Figure 2:   Experiment Results Summary for FFmpeg

Similarly, the experiment results for Outside In, at 500,000 tests with both features enabled, yielded an average of 78 unique crashes versus an average of 65 unique crashes for the equiprobable scenario (20% improvement). By 1 million tests, the score was 113 versus 70, respectively (61% improvement).

Due to the shorter duration of the Outside In campaign, we did not reach 5 million tests. The mixed scenarios produced between 84 and 104 unique crashes after 1 million tests. Results are summarized in Table 3.

Table 3:   Experiment Results Summary for Outside In

Outside In	Unique Crashes Found (Average of 5 Runs Each)			
N Trials	S0 R0	S0 R1	S1 R0	S1 R1
500	0	0	0	0
1K	0	0	0	0
5K	1	2	1	1
10K	3	3	3	3
50K	14	14	15	16
100K	24	22	26	28
500K	65	56	75	78
1M	70	84	104	113

Figure 3 illustrates that even with much fewer tests run, there is a clear advantage to using a selection algorithm.
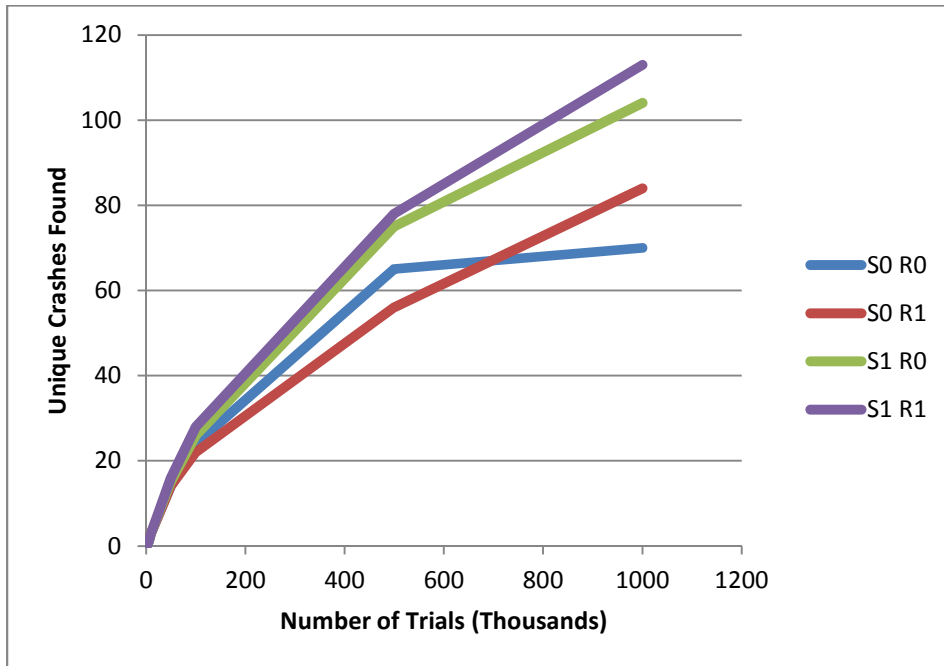
Figure 3: Experiment Results Summary for Outside In

As shown in Tables 2 and 3 as well as Figures 2 and 3, the longer the campaigns ran, the larger the disparity between the purely random (S0R0) and learning (S1R1) scenarios.

6 Limitations and Future Work

In this section, we discuss the limitations of this work and explore possible improvements and extensions.

We observed that given similar elapsed times (18 days for FFmpeg and 6 days for Outside In), the S1R1 instances completed about 10% fewer total iterations than the S0R0 instances. While we did not collect granular timing data and therefore did not perform a detailed analysis, one possible explanation of this difference is because S1R1 found more unique crashes, more time was spent performing the additional analyses that the CERT BFF does for each unique crash.[2]

Furthermore, by focusing on parameters that produced more unique crashes, we expect the S1R1 campaigns to observe more duplicate crashes as well. This feature implies more time spent determining uniqueness and less time spent fuzzing. While we have explained an intuitive rationale for this phenomenon, future work could include a detailed timing analysis of fuzz campaigns to confirm our conclusions.

While the team continually tunes the crash uniqueness algorithm as we test various types of applications, we learned from coordinating results with software vendors that the unique fuzzy stack hashes do not necessarily correspond directly to unique application bugs. Improving the fuzzy stack hashing algorithm and augmenting it with other types of analysis could strengthen data used to compare automated bug-finding techniques, both in the CERT BFF and other testing frameworks.

The controlled test described in this report involved executing fuzz campaigns against two applications for several days each. The results presented are consistent with what we observed in the ongoing operational use of the CERT BFF against various software applications. However, the data set used to draw conclusions in this report could be strengthened by testing additional applications in a controlled manner.

While the algorithm in this report proved to be effective, we have begun exploring other statistical approaches that may lend themselves more directly to analysis and scaling, such as Bayesian inference [25]. We leave exploration of these approaches to future work.

---

[2]   When the CERT BFF finds a new unique crash, it runs the test case through the GNU debugger (GDB) and the valgrind and callgrind tools and records the stderr produced for the test case.

7 Conclusion

In this report, we described a workflow for black-box fuzz testing and an algorithm for selecting fuzz parameters to maximize the number of unique application errors discovered during a fuzzing campaign. We presented an open-source implementation of the algorithm in the CERT BFF, which was used to find several previously unknown security vulnerabilities.

Further, we described a new methodology for testing the algorithm and discussed the results. We found that, after an initial learning period, the algorithm significantly improved the efficiency of discovering unique application errors over basic parameter selection techniques.

We have shown that applying the simple machine-learning algorithm presented here can markedly improve black-box fuzzing techniques. The algorithm described in this report continues to be used to discover new security vulnerabilities in software applications. The source code for a working implementation of the algorithm is available in *Appendix B: Scorable Set Example Source Code (Python)*. Also, the algorithm described here was implemented in the CERT BFF version 2.5 and is available under a liberal software license compatible with GPLv2 at http://www.cert.org/download/bff.

Appendix A: CERT BFF Configuration

This appendix contains the configuration used for the CERT BFF instances used in the FFmpeg campaigns described in this report. Each instance of a given scenario was configured with a different start_seed value (e.g., 0, 10000000, 20000000)

```
[campaign]
id=Default BFF Campaign
distributed=True

[target]
cmdline=ffmpeg -y -i $SEEDFILE -acodec pcm_s16le -f rawvideo /dev/null
killprocname=ffmpeg

[directories]
remote_dir=~/remote
seedfile_origin_dir=%(remote_dir)s/seeds
debugger_template_dir=%(remote_dir)s/debuggers/templates
output_dir=~/results
crashers_dir=%(output_dir)s/crashers
seedfile_output_dir=%(output_dir)s/seeds
local_dir=~/fuzzing
seedfile_local_dir=%(local_dir)s/seeds
cached_objects_dir=%(local_dir)s
temp_working_dir=%(local_dir)s/tmp
watchdog_file=/tmp/bff_watchdog

[zzuf]
copymode=0
start_seed=0
seed_interval=500
max_seed=10000000000

[verifier]
backtracelevels=5
savefailedasserts=0
minimizecrashers=False
minimize_to_string = False

[timeouts]
zzuftimeout=3
killproctimeout=60
progtimeout=5
debugger_timeout=15
valgrindtimeout=60
watchdogtimeout=3600

[memcache]
server=127.0.0.1
port=11211
```

For the Outside In campaigns, a similar configuration file was used that substituted the previous [target] section with the following:

```
[target]
cmdline=exsimple $SEEDFILE /dev/null
killprocname=exsimple
```

Appendix B: Scorable Set Example Source Code (Python)

This appendix contains a simplified implementation of the algorithm described in this report.

```python
import random
import scipy.stats
from collections import defaultdict

class ScorableSet(object):
  def __init__(self):
    self.items = {}
    self.successes = defaultdict(int)
    self.tries = defaultdict(int)
    self.confidence = 0.95
    self.scaled_scores = defaultdict(float)
    self.lambdas = defaultdict(float)
    self.sum_scores = 0.0
    self.probabilities = defaultdict(float)

  def add_item(self, key, value):
    self.items[key] = value

  def record_success(self, key, n=1):
    self.successes[key] += n

  def record_tries(self, key, n=1):
    self.tries[key] += n

  def update_probabilities(self):
    self.scaled_scores = {}
    self.sum_scores = 0.0

    # calculate scores for items
    for k in self.items.keys():
      if not self.tries[k]:
        # we have no tries, so successes don't matter yet
        self.scaled_scores[k] = 1.0
      else:
        # calculate the upper confidence bound for this item
        self.lambdas[k] = float(self.successes[k]) / float(self.tries[k])
        chisq = 1.0 - self.confidence
        df = 2 * (self.successes[k] + 1)
        delta = (scipy.stats.chisqprob(chisq, df) / 2) / self.tries[k]
        self.scaled_scores[k] = self.lambdas[k] + delta
      self.sum_scores += self.scaled_scores[k]

    # update the probabilities
    self.probabilities = {}
    for k in self.scaled_scores.keys():
      self.probabilities[k] = self.scaled_scores[k] / self.sum_scores
```

```python
  def next_key(self):
    self.update_probabilities()

    # pick the next key
    x = random.uniform(0, 1)
    cumulative_probability = 0.0
    for (k, p) in self.probabilities.items():
      cumulative_probability += p
      if x < cumulative_probability:
        return k

def printstats(s):
  print '====='
  for k in 'ABCDE':
    print "Item:%s successes=%2d tries=%2d lambda=%0.3f ucb=%0.3f p=%0.3f" %
(k, s.successes[k], s.tries[k], s.lambdas[k], s.scaled_scores[k],
s.probabilities[k])

if __name__ == '__main__':
  '''Simulates a fuzzing campaign in which there are 5 files A-E. File A has
a crash
  density of 0.2, File B a crash density of 0.33, and File C a crash density
of 0.14.
  Each iteration will print the current number of successes, tries, and
probability
  that file will be chosen.'''
  print "Scorable Set Demo"
  s = ScorableSet()
  for (key, value) in zip('ABCDE', '12345'):
    s.add_item(key, value)

  for x in xrange(500):
    k = s.next_key()
    printstats(s)
    if k == 'A' and not (x % 5):
      s.record_success(k)
    elif k == 'B' and not (x % 3):
      s.record_success(k)
    elif k == 'C' and not (x % 7):
      s.record_success(k)
    s.record_tries(k)
```

References

URLs are valid as of the publication date of this document.

[1]   B. P. Miller, L. Fredriksen, and B. So. (1990, December). An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*. [Online]. 33(12), pp. 32-44. Available: http://dl.acm.org/citation.cfm?id=96279 [Accessed 22 August 2012].

[2]   B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. (1995, October). *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. [Online]. Available: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf [Accessed 22 August 2012].

[3]   J. E. Forrester, B. P. Miller, and USENIX Association, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium*—Volume 4, Seattle, WA, 2000.

[4]   P. Godefroid, M. Y. Levin, and D. Molnar. "Automated Whitebox Fuzz Testing," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2008.

[5]   D. Molnar, X. C. Li, D. A. Wagner, and USENIX Association, "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, Montreal, QC, 2009, pp. 67-82.

[6]   H. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP Fuzzer," in *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*, New York, NY, 2007, pp. 47-56.

[7]   P. Godefroid, *From Blackbox Fuzzing to Whitebox Fuzzing towards Verification*, presented at the International Symposium on Software Testing and Analysis (ISSTA) 2010, Trento, Italy, 2010, pp. 1-38. [Online]. Available: http://research.microsoft.com/en-us/um/people/pg/public_psfiles/talk-issta2010.pdf [Accessed 22 August 2012].

[8]   J. Foote. (2011, December 9). *JasPer memory corruption vulnerabilities* [Online]. Available: http://www.kb.cert.org/vuls/id/887409 [Accessed 22 August 2012].

[9]   W. Dormann. (2009, September 5). *VMware VMnc AVI video codec image height heap overflow* [Online]. Available: http://www.kb.cert.org/vuls/id/444513 [Accessed 22 August 2012].

[10]   W. Dormann. (2012, January 12). *Microsoft Indeo video codecs contain multiple vulnerabilities* [Online]. Available: http://www.kb.cert.org/vuls/id/228561 [Accessed 22 August 2012].

[11] W. Dormann. (2012, January 12) *Adobe Flash ActionScript AVM2 newfunction vulnerability* [Online]. Available: http://www.kb.cert.org/vuls/id/486225 [Accessed 22 August 2012].

[12] M. Aslani, N. Chung, J. Doherty, N. Stockman, and W. Quach, "Comparison of Blackbox and Whitebox Fuzzers in Finding Software Bugs," presented at the *Team for Research in Ubiquitous Secure Technology (TRUST) Autumn 2008 Conference*, Nashville, TN, 2008.

[13] FFmpeg Developers. (2012, January 11). *FFmpeg.org* [Online]. Available: http://ffmpeg.org/ [Accessed 22 August 2012].

[14] Microsoft Corporation. (2011, June 3). *Microsoft Secure Development Lifecycle (SDL) Process Guidance Version 5.1* [Online]. Available: http://www.microsoft.com/downloads/details.aspx?FamilyID=E5FF2F9D-7E72-485A-9EC0-5D6D076A8807&displaylang=en [Accessed 22 August 2012].

[15] CERT. (2011, February 28). *CERT Basic Fuzzing Framework (BFF) v2.5* [Online]. Available: http://www.cert.org/download/bff/index.html [Accessed 22 August 2012].

[16] S. Hocevar, "zzuf - multiple purpose fuzzer," presented at the Free and Open Source Software Developers' European Meeting (FOSDEM), Brussels, Belgium, 2007.

[17] S. Hocevar. (2011, May 12). *zzuf - multi-purpose fuzzer* [Online]. Available: http://caca.zoy.org/wiki/zzuf [Accessed 22 August 2012].

[18] H. Dai, C. Murphy and G. Kaiser, "Configuration Fuzzing for Software Vulnerability Detection," in *ARES '10 International Conference on Availability, Reliability, and Security*, Krakow, 2010, pp. 525-530.

[19] Microsoft Security Engineering Center (MSEC). (2009, June 17). *!exploitable Crash Analyzer - MSEC Debugger Extensions* [Online]. Available: http://msecdbg.codeplex.com/ [Accessed 22 August 2012].

[20] S. Bekrar, C. Bekrar, R. Groz, L. Mounier, and IEEE, "Finding Software Vulnerabilities by Smart Fuzzing," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation*, Berlin, 2011.

[21] T. Wang, T. Wei, G. Gu and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *2010 IEEE Symposium on Security and Privacy (SP)*, Oakland, CA, 2010.

[22] M. Eddington. (2011, April 5). *Peach Fuzzing Platform* [Online]. Available: http://peachfuzzer.com/ [Accessed 22 August 2012].

[23] J. Holland, *Adaptation In Natural And Artificial Systems: An Introductory Analysis With Applications To Biology, Control, And Artificial Intelligence*, Cambridge, MA: MIT Press, 1992.

[24] Oracle. (2012, January 12). *Oracle Outside-In Technology* [Online]. Available: http://www.oracle.com/us/technologies/embedded/025613.htm [Accessed 22 August 2012].

[25] A. Gelman, J. Carlin, H. Stern, and D. Rubin, *Bayesian Data Analysis, Second Edition*, Boca Raton, FL: Chapman & Hall/CRC Texts in Statistical Science, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE August 2012	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Probability-Based Parameter Selection for Black-Box Fuzz Testing	5. FUNDING NUMBERS FA8721-05-C-0003

6. AUTHOR(S)
Allen D. Householder and Jonathan M. Foote

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2012-TN-019

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) ESC/CAA 20 Schilling Circle, Building 1305, 3rd Floor Hanscom AFB, MA 01731-2125	10. SPONSORING/MONITORING AGENCY REPORT NUMBER n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS	12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

Dynamic, randomized-input functional testing, or black-box fuzz testing, is an effective technique for finding security vulnerabilities in software applications. Parameters for an invocation of black-box fuzz testing generally include known-good input to use as a basis for randomization (i.e., a seed file) and a specification of how much of the seed file to randomize (i.e., the range).This report describes an algorithm that applies basic statistical theory to the parameter selection problem and automates selection of seed files and ranges. This algorithm was implemented in an open-source, file-interface testing tool and was used to find and mitigate vulnerabilities in several software applications. This report generalizes the parameter selection problem, explains the algorithm, and analyzes empirical data collected from the implementation. Results of using the algorithm show a marked improvement in the efficiency of discovering unique application errors over basic parameter selection techniques.

14. SUBJECT TERMS Security, verification	15. NUMBER OF PAGES 30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL