Introduction to Software Security
Chapter 7.1: Introduction to Fuzz Testing

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 2.0, January 2022.

Objectives

- Learn what is fuzz testing
- Understand what it is used for
- Understand the different forms of fuzz testing techniques
- Motivate our detailed study of classic and modern fuzz testing

What is Fuzz Testing?

It is well known that reliability is the foundation of security. So thorough testing is an important way to improve security by reducing bugs. However, extensive testing is a lot of work, and most software is not nearly as well tested as it could be. Fuzz testing is a technique that supplements general software testing, often uncovers security bugs, and most importantly, requires relatively little effort to expand test coverage and get useful results.

Fuzz testing explores the state space of your program by probing it with random inputs called “fuzz” and observing the results, checking for unexpected results that suggest the presence of underlying bugs. Since the input is random, it is easy to create many test cases, though without the intelligent focus that human-created tests would have.

You can think of fuzz testing as the automated version of monkeys-at-the-keyboard, which you can imagine would give almost any software fits. Since fuzz testing is conducted without targeting any specific program logic the way conventional tests do, the other key insight behind it is that the test result depends on an easily observable class of bugs.

How it Works

All software testing consists of invoking the target code and then examining the resulting behavior with a determination that the test either passed or failed. The specific method that a test uses to make this determination is known as an *oracle*. In the simplest case, a test of a sales tax computation function might request the program to compute a 5% tax rate on $100, and use an oracle of “result = $5”. Each test case requires an oracle to decide if the test passed or failed. Fuzz testing differs from most other forms of testing in that it uses a simple, almost simplistic, oracle that passes the test if it does not crash or hang. The crash or hang cases are far from complete. For example, if you compiled a C program and, instead of an object file containing Pentium instructions, you got a copy of the Gettysburg Address in Latin, that would, to a human, obviously be an incorrect result. However, in fuzz testing, since the compiler did not crash or hang, this output would not be considered a failure. However, the crash or hang determination is easy to implement and exposes a surprising number of important bugs.

Fuzz testing pummels its target with unexpected inputs, searching for input that causes crashing or hanging. When failures are detected, it tends to be due to specific classes of bugs such as:

- Unchecked conditions (buffer overruns)
- Unexpected combinations (state inconsistencies)
- Unexpected input values (range or value errors)

If any of these classes of bugs happen, that usually indicates that the input managed to own bits in the program state that causes unexpected behavior. Fuzz testing cannot distinguish exploitable security bugs from the rest, but it does find interesting inputs that trigger bugs that would be otherwise difficult to uncover. These bugs can be considered potentially exploitable, though it often requires human analysis to know for sure if there is security exposure. Often it is easiest to just fix the bug without bothering to do the difficult investigation to determine exploitability.

Complementing software testing

Software testing is a huge field that has been studied since the earliest days of computing. Typically, the goal is to check that a known input generates the expected output, as determined by a simple oracle. One important measure of the effectiveness of a test suite is *code coverage,* the degree to which tests execute all the branches within the target code. Note that code coverage measurements can be assessed at different levels of granularity such as for functions, statements, branch cases (e.g., both the *then* and *else* clauses), basic blocks, or even machine language instructions.

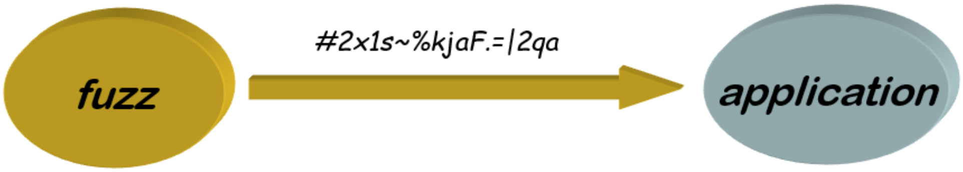
A good test suite should provide high levels of code coverage, but it will probably fall well short of 100%. Fuzz testing often has a knack for filling in the cracks because the large number of different fuzz inputs and unexpected patterns of input tend to find the exceptional cases that conventional testing misses. While fuzz testing cannot replace the need for basic software testing, the use of both kinds of test techniques combine to achieve better code coverage.

Broadly speaking, software testing can be applied at different levels. Unit tests target specific chunks of code at the smallest scope, often substituting fake dependencies so as to focus the test as tightly as possible. Integration tests exercise multiple components to check how they interact and work together, and end-to-end tests at the largest scale check that complete systems perform correctly. Fuzz testing can be applied to any of these levels of tests to best complement other testing.

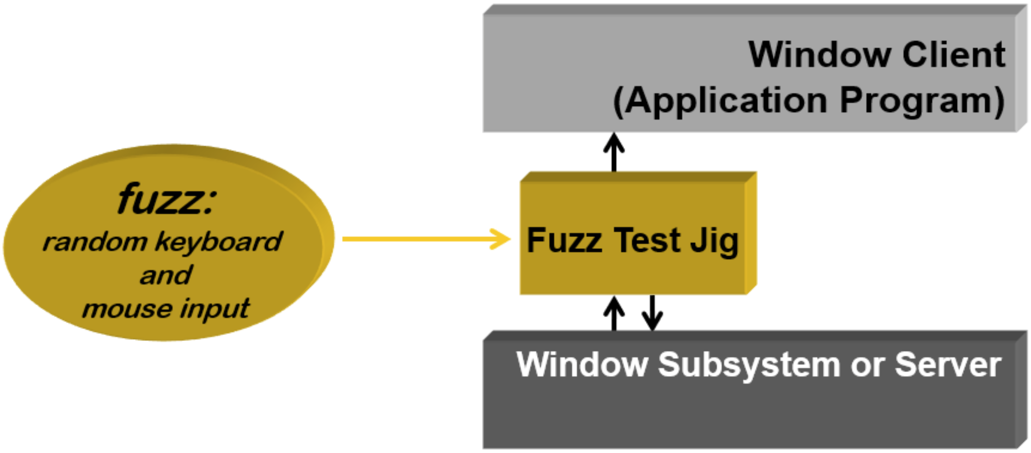
Forms of Fuzz Testing

The simplest form of fuzz testing is exemplified by Unix command line programs that read standard input and produce standard output. Fuzz testing of these programs consists of feeding themt the fuzz data as input, ignoring any output, and checking if the programs crashes or hangs. For programs that accept input from multiple files, separate test files containing the fuzz data can be prepared as input, and tested in a similar manner.

fuzz | application



GUI applications are fundamentally different because their main source of input is from a window-based user interface, typically consisting of a sequence of events from the mouse (movements and clicks) and keyboard (key up or down). The main output of these applications consists of screen drawing, though this is usually ignored for fuzz testing since the focus is on detecting hangs and crashes. Fuzz testing GUI of applications requires a suitable test harness that presents fuzz data as a stream of events (such as a gang of monkeys might produce), and consuming screen drawing requests (since there is no point actually writing on a screen which we are ignoring anyway). In addition to testing GUI applications, this technique can be used to test the window manager event queuing and processing layer of the system. Fuzz testing GUI applications you have a choice of only fuzzing with well-formed events (e.g. key up events only follow corresponding key down), or when necessary also testing invalid or “impossible” events as well (e.g. multiple key up events for the same key without any intervening key down, mouse movements at high speed or greater distance than any screen dimension).



Testing phone applications is similar in many ways to testing GUI-based applications. Instead of a keyboard and mouse, input is generated primarily by touch gestures on the screen, along with voice and camera input. Similar to other testing scenarios, there needs to be a way for the testing program to inject input into the phone. On Android, for example, the Android debugger (adb) can be used in conjunction with the Monkey test generation program.

Network services, including web servers, represent another class of applications that require a test harness. In this case, fuzz data needs to be presented to the application in the form of network input, such as a network packet. The server response is consumed as the test proceeds looking for hangs and crashes.

These categories of fuzz testing cover many types of programs but are by no means complete. If you wanted to test a new category of program, such as in an embedded device controller, you would have to be able to identify the input mechanisms and develop a test harness. The choice of test harness corresponds to what layer of abstraction you want to perform the fuzz testing. For example, when you test a Web server application by driving it with fuzz data as HTTP requests, that represents a choice not to exercise the actual transmission of Web traffic, because testing that part of the system is not necessary.

Multithreaded applications are a great opportunity to use fuzz testing because the vicissitudes of thread scheduling can be difficult to test thoroughly. Fuzz testing can randomize thread scheduling by orchestrating when threads preempt and block, including unusual patterns that should work but may only rarely happen in the wild.

Fuzz Variations

Fuzz testing is capable of dynamically generating many penetrating test inputs at high speed, but the entire search space of possible inputs is almost always so vast that we cannot begin to cover it by brute force alone. For this reason, we must be strategic and make choices in order to achieve the best chance of finding new bugs for the effort and runtime cycles spent.

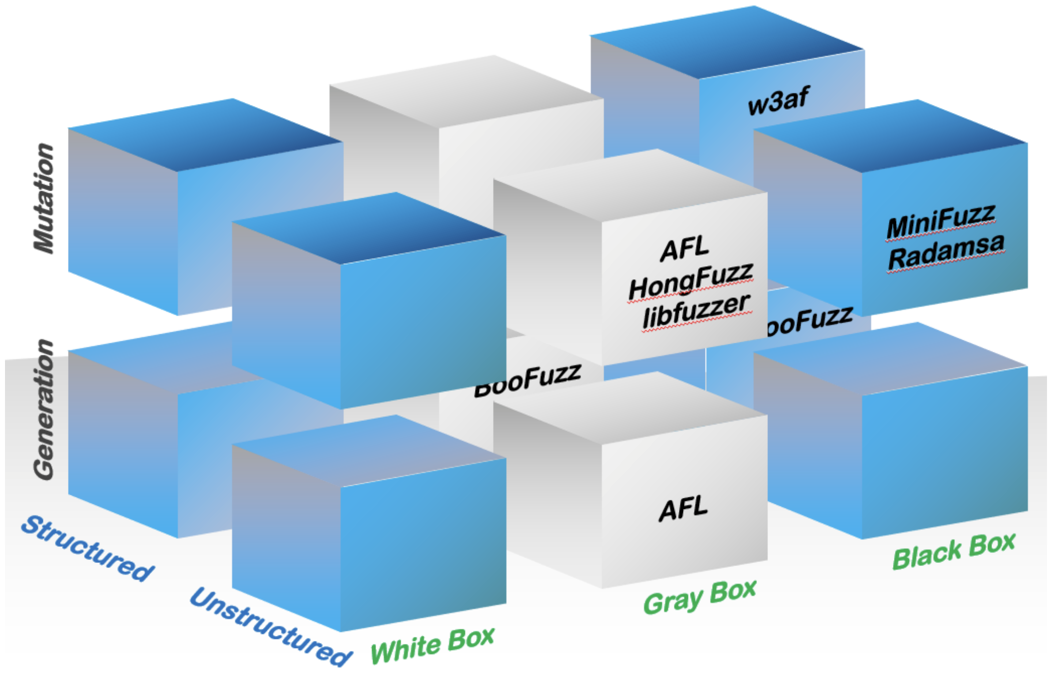
Modern fuzz testing strategies have been divided into three categories:

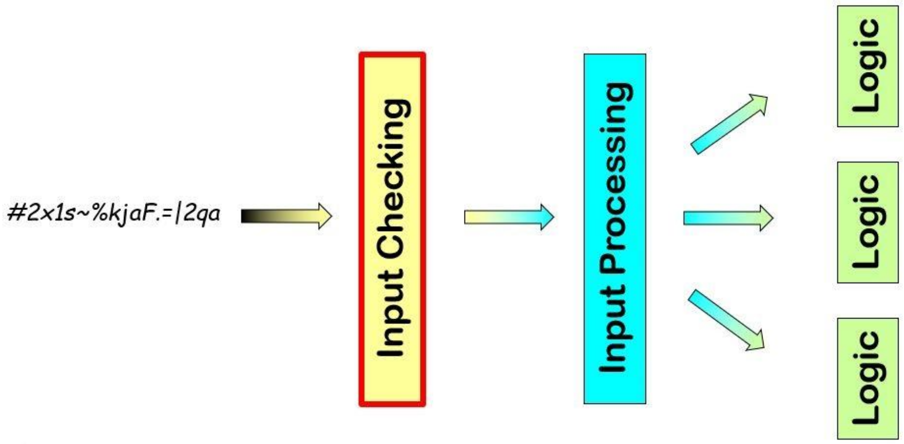
- Unstructured vs. structured input
- Generation vs. mutation
- Black vs. gray vs. white box

In the following subsections we describe each of these categories and how to use it.

Unstructured vs. Structured Input

The simplest form of random testing is a random stream of bytes. This stream might have some simple constraints on the byte values, such as not including the null (zero) or newline characters, but otherwise has no internal structure. While this *unstructured* input has demonstrated the ability to cause crashes or hangs in real programs, it does have its limits. Such unstructured input is less likely to exercise paths deep within the program’s control flow.





Programs are often designed with multiple layers of input processing. The first layer typically checks the input for valid format. For packet or message based programs, that means that the input is of the correct length, has valid fields, and perhaps has a valid checksum. For programs that accept strings as input, then the check might be for length, valid character set, and contains a valid command.

Unstructured input will often trigger errors in this first input-checking layer. However, it sometimes surprisingly generates input that propagates deeper into a program's logic. Fuzz testing can try to improve this situation by generating input that conforms to a *structure* that is likely to be recognized by the program being tested.

Command lines: Partial commands that get past syntax checking and into the program's logic. For example, if you were testing a web server, you might start the input line with a randomly chosen but valid HTTP request method such as "GET" or "POST".

Further structuring might include randomly chosen but valid parameters and input values that have random values but are of the correct type (such as numeric).

Window systems: Fuzzing with valid keyboard and mouse events targets the main type of input used by GUI applications. So such random input will conform to the message format for these input events.

Further structuring might constrain the input in various ways such as guaranteeing that a given key-down event is matched with a corresponding key-up event. In addition, event values, such as the X-Y coordinates of the mouse would be constrained to be within the bounds of the window.

Network protocols: Properly formed TCP/IP packets (or higher level protocols) with random values in the fields are more likely to exercise network stack implementations effectively.

Compilers: Input constrained to contain valid input tokens such as keywords, operators, identifiers and constants.

The input could be further structured to follow the language grammar, therefore be valid syntactically (though not necessarily semantically meaningful).

There is an important tradeoff when structuring input. As we have observed, structured input can test deeper into a program's structure. On the other hand, generating structured input requires the creation of a custom fuzz generator that is aware of the constraints on the input. Creating such a custom fuzz generator is significantly more work than simply using an existing unstructured generator.

If you are involved in a software project for an extended period of time, then the investment of effort in such a structured generator can be worthwhile. However, if you are testing a wide variety of software, then such an effort might be prohibitive.

Generation vs. Mutation Testing

The most basic way to generate fuzz data is to create it from scratch, such as was done in the original fuzz testing. Each time that you run the fuzz tool, a new random input stream is generated. Such *generational* fuzz testing applies to both unstructured and structured input.

As we mentioned above, adding some structure to the test input can allow for testing deeper into the program's logic. However, writing a custom structured fuzz generator for each program is a lot of work. An alternative approach is to use *mutational* input generation. The idea behind mutation testing is that you start with one or more existing test inputs, called *seeds,* and then incrementally modify that input to generate a new one. The modification might be randomly choosing a byte or word in the existing seed input and changing it to a random value. Other modifications might include randomly adding a value to a byte or word, flipping bits in a byte or word, and deleting or duplicating bytes or words.

Starting with one valid input can then generate many new inputs. The valid input might be a valid SQL query or program code or command line or image file. Of course, starting with a valid input and then randomly transforming it may not generate another valid input, but often will. Such modifications provide an interesting collection of inputs, both valid and invalid, potentially testing many parts of the code.

A note on random values:
Fuzz data generation should use pseudo-random values to control the choice of specific data values produced. There is no need to use cryptographically secure random numbers because there is no adversarial aspect to this testing itself, although it very well may find bugs that adversaries might discover someday. Another detail of random number generation for fuzz testing to consider is using a *random seed*. For pseudo-random number generators that support seeds, the subsequent sequence of values will be completely determined by the seed value. Setting and recording the seed makes it easy to reliably reproduce the fuzz data exactly which makes investigation of bugs found far more convenient since the result might depend on a subtle detail of the fuzz data used. On the other hand, repeated fuzz testing sessions with different random input each time potentially allows more of the program state space to get coverage, also without knowing the seed or saving copies of the relevant fuzz data it may be challenging to reproduce the results.

Black, Gray and White Box Testing

The original fuzz testing was based on using no knowledge about the structure of the program that it was testing. This strategy, called *black box* testing, is simple to implement.

However, people experienced with testing have noted that if you know something about the program that you are testing and the effect of the input on how the program executed, you might be able to create new inputs that can test more of a program's logic.

Researchers have suggested using detailed knowledge about the structure of the program, such as its control flow graph and data dependence calculations. For example, if an input tested the `then` clause of an `if`-statement, they would try to generate an input subsequently tested the `else` clause[1]. Testing that uses this knowledge of the program's structure is called *white box* testing. While potentially quite powerful, such testing requires a sophisticated code analysis framework and proof system. It has been quite slow, resulting in no commonly used tools based on this approach.

The most common approach for using information about a program is to track which blocks of code in the program executed for a given input. An approach such as this one, that uses limited information about the program's structure is called *gray box* testing. Gray box testing uses limited program knowledge to get some of the advantages of white box testing while still maintaining reasonable performance. While more complicated to build than a black box tester, it is significantly easier to build and has lower execution overheads than white box testing.

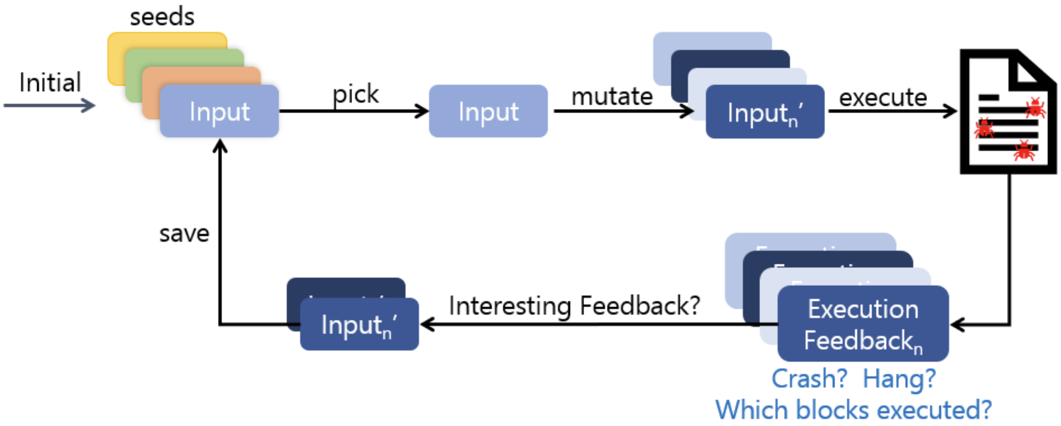
[1] For example, see: Patrice Godefroid, Michael Y. Levin and David Molnar, "Automated Whitebox Fuzz Testing", *16th Annual Network & Distributed System Security Symposium (NDSS),* San Diego, California, February 2008.

The original and simplest approach to fuzz testing was to combine black box with generational testing. While such testing is simple, it is the least likely to test deeply into the program's structure, though it sometimes gives surprisingly good results.

Mutational Gray Box Testing (Coverage Guided Testing)

Mutational gray box testing, also known as *coverage guided testing,* combines to form a powerful and flexible approach that has proven effective for a wide variety of applications. The tester will then generate new inputs by modifying a previous one. It uses information about which paths in the program have executed to determine which inputs to modify based on its estimate of which are likely to explore new paths in the program. Fuzz testers, such as AFL (American Fuzz Lop), libFuzz and HongFuzz, use this approach.

We start with valid input test data, which should already be available from existing software test cases, and then mutate that in order to expand its code coverage potential. The key concept here is to use the results of each test as feedback to drive further mutation, aiming to incrementally penetrate deeper into the code to increase coverage.



Let's take a look at how this works in more detail. The first thing we do is compile the program with a special version of the compiler (such as afl-clang). This special compiler will insert instrumentation code into the program to record which blocks executed when the program ran.

We start the actual testing with a set of test inputs called "seeds" (not to be confused with random number generator seeds). The test driver picks one of the seeds at random, removing it from the collection, mutates it randomly according to a set of rules, then uses it as fuzz data for a test run. Each test run executes with instrumented code that records which blocks in the program were executed.

A test run will either crash, hang, or complete normally. In any case, the program will have executed some of its basic blocks and record this execution information in a file. If the program crashed or hung, then the execution is considered interesting, and the input and execution information is recorded. If the program completed normally but happened to have executed some new basic blocks previously unreached, then that is also considered interesting. Test inputs that produce interesting results are saved in the collection of seeds for future mutation. Uninteresting results do not do this, effectively cutting off further testing with this seed. In this way the fuzz testing focuses on the more interesting seeds and their mutations, while defocusing fuzz data inputs that are likely to exercise the same basic blocks over and over.

Tools such as AFL (the subject of Module 7.3), libFuzzer, and HongFuzz work in this way, and are a good first choice for fuzz testing of many applications. You can start fuzz testing with these tools and a modest set of seeds and probably get some interesting results. Based on how much code coverage was accomplished, you can enhance the seed test inputs over time in order to touch other important parts of the code that might have been missed. By repeating this process with a little knowledge of the target code you can incrementally grow code coverage quickly to a respectable percentage.

Summary

Fuzz testing is a simple technique that every programmer should have in their toolkit.

- Learned about what is fuzz testing
- Discussed how it works and what it is used for
- Discussed the different forms of fuzz testing techniques
- Learned about coverage guided testing, which is gray box mutation testing
- Motivated our detailed study of classic and modern fuzz testing

Exercises

1. Write a simple fuzz test for the command line and run it on a few simple commands that take standard input. Hint: for Unix you can use `/dev/urandom` as a source of random bytes. Warning: for safety, avoid using commands that might cause damage, such as `rm`.
2. Write a simple program that reads standard input and writes standard output that contains an intentional bug. For example, the program might output lines with an even number of characters twice, or three times for odd length, but lines that begin with “!” an infinite number of times. Fuzz test your program with your testing from Exercise 1 to find the bug.
3. Think of new types of applications to apply structured input data for fuzz testing, beyond those listed in the text. For each application:
    a. Describe the input to that application and how it is structured.
    b. What part of the input would you structure and where would you apply the randomness?
    c. How would this structured fuzz data affect the parts of the code tested in this application?
4. Describe how a mutational gray box tester uses feedback in the form of blocks executed in the code. What is the goal of the mutation and how does the feedback inform the testing process?
5. Challenging: How would you extend fuzz testing of GUI or phone applications to include camera and microphone inputs? What kind of test harness would you need to accomplish this kind of testing?

6. Try out the hands-on exercises associated with the fuzz testing module: https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Exercises/7.1_Fuzz_Testing_Exercise.html