Chapter 38
Introduction to Fuzz Testing

Revision 3.3, December 2025.

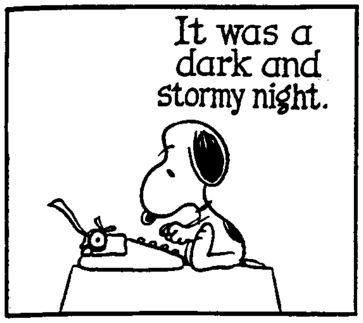
Objectives

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

38.1 A Bit of History

Fuzz testing started on a dark and stormy night. It really did.

On a stormy night in the early fall of 1988, one of this book's authors was logged on from home to the UNIX system in his office on campus. He was dialed in to the UNIX system in his computer via a 2400 baud[1] acoustic coupler modem. That night, there was thunder, lightning, and heavy rain. As a result, there was noise on the phone line, the kind of noise that sounds like crackling or static when you listing to it.



Importantly, the 2400 baud modem technology did not yet have error correction, so noise on the phone line caused garbage characters to appear. When he was typing characters, it was a race to hit "return" before he got garbage characters in the command or input that he was typing. This was not surprising to anyone of that generation who used these non-error correcting modems. What was surprising was that the commands that read the garbage characters were crashing. And significant programs – like text editors, word processors, and network utilities – were crashing.

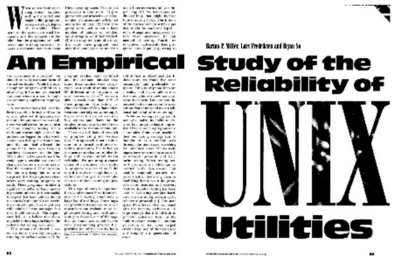
As a result, he decided to get some of his students to investigate this behavior to better understand its cause. That semester, fall 1998, Miller was teaching the graduate Advanced Operating System class (CS736) at the University of Wisconsin-Madison. In the class the student read a large collection of foundational papers from the operating systems literature and do a semester-

[1] Approximately bits per second.

long research project. Miller included fuzz testing as one of the options for that semester. You can see the original class handout in Figure *1*. Three teams attempted the project and only one team succeeded, Lars Fredricksen and Bryan So. The main result of their efforts was that they were able to crash between 25% - 33% of the utility programs that they tested on a variety of UNIX systems (more about these results in the next chapter).

It was already interesting that they could crash so many programs. However, what made this work impactful was they did not stop there. For each crash, they debugged it and identified the underlying cause. They then grouped these into categories to try to understand the behaviors that were having the biggest impact on software reliability.

Getting the results published was a surprisingly challenging process. From the class project results, we put together a research paper and submitted it to a distinguished computer science journal. The reviews from that first submission were brutal, including one that suggested that the faculty member consider leaving the field of computer science (though the review was not worded that politely). This research did not fit the model of testing and software engineering papers of the day since it did not have enough formal prose or equations, so was considered a poor effort at best. However, the editor of the journal *Communications of the ACM* was more farseeing, found reviewers who were less entrenched, and accepted it for publication in 1990. What started as a class project has now become the foundation for a global phenomenon.

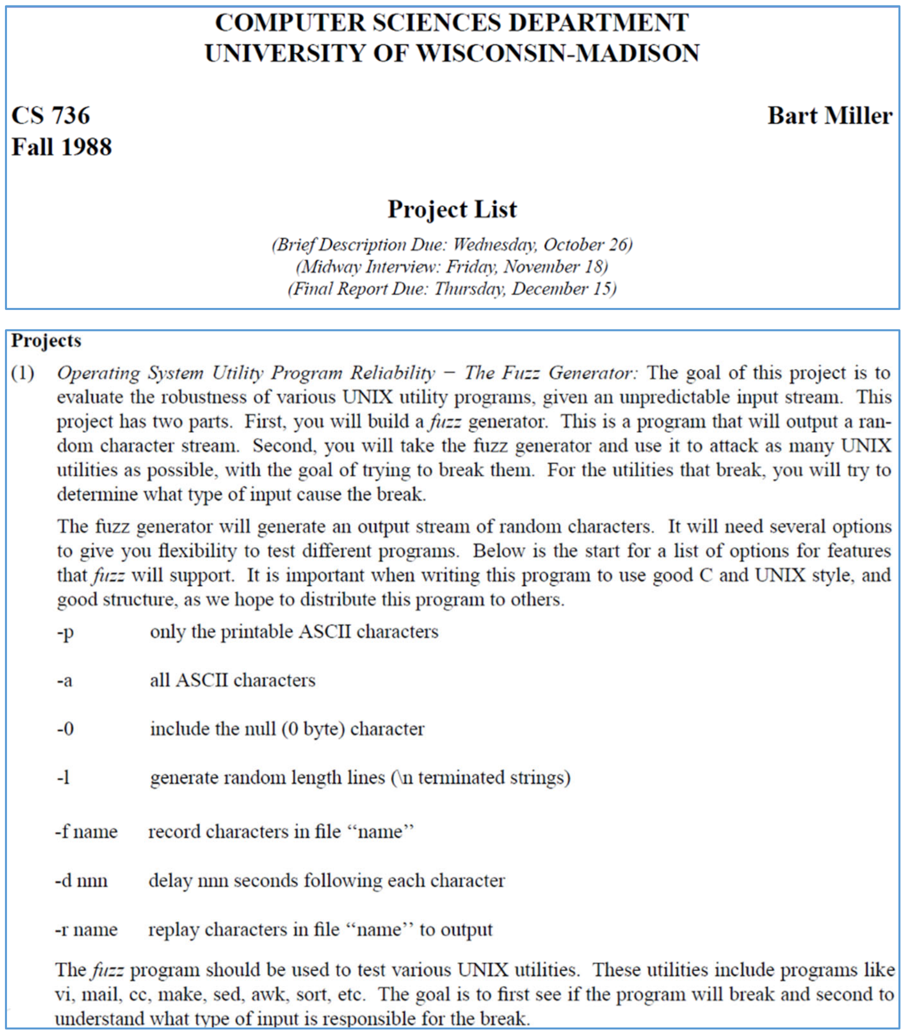

Barton P. Miller, Lars Fredriksen and Bryan So

An Empirical Study of the Reliability of UNIX Utilities

38.2 Background

In its simplest form, fuzz testing, or "fuzzing" as it is commonly called now, is just feeding random input to a program and see if it crashes or hangs (stops responding to input). You can think of it as randomly exploring the program's state space to see if you can cause any unexpected behaviors. These behaviors might include reading or writing outside the valid range of a data structure, causing the program's internal variables to be in an inconsistent state, or perform calculations on values that exceed the expected range.

Besides finding bugs, this technique has been important to security analysts. If you cause a program to crash, you have effectively caused a state transition within the program that was not anticipated by the programmer. You are wandering into undefined regions of the program's state space, accessing code or data in a way that the program was not designed to handle.



COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

CS 736 **Bart Miller**
Fall 1988

Project List

(Brief Description Due: Wednesday, October 26)
(Midway Interview: Friday, November 18)
(Final Report Due: Thursday, December 15)

Projects

(1) *Operating System Utility Program Reliability − The Fuzz Generator:* The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. This project has two parts. First, you will build a *fuzz* generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them. For the utilities that break, you will try to determine what type of input cause the break.

The fuzz generator will generate an output stream of random characters. It will need several options to give you flexibility to test different programs. Below is the start for a list of options for features that *fuzz* will support. It is important when writing this program to use good C and UNIX style, and good structure, as we hope to distribute this program to others.

-p	only the printable ASCII characters
-a	all ASCII characters
-0	include the null (0 byte) character
-l	generate random length lines (\n terminated strings)
-f name	record characters in file "name"
-d nnn	delay nnn seconds following each character
-r name	replay characters in file "name" to output

The *fuzz* program should be used to test various UNIX utilities. These utilities include programs like vi, mail, cc, make, sed, awk, sort, etc. The goal is to first see if the program will break and second to understand what type of input is responsible for the break.

Figure 1: The 1988 Class Assignment that Started Fuzz Testing

In effect, using the terminology from Chapter 3, you are owning bits that you were not supposed to own.

In software testing, when you run the program with a test input, you need something to tell you whether the output of the program was correct or not for that input. The thing that tells you whether an input is correct is called an *oracle*. In general, automatically understanding what is correct output can be quite complex and difficult to do.

Fuzz testing has a simple, almost simplistic oracle: the program is considered to be correct if it does not crash or hang. If it does crash or hang, that is considered a failure. For example, if the Java compiler output the Gettysburg Address in Latin instead of Java byte code, but did not crash, that would be a correct result relative to fuzz testing. However, clearly the compiler writers would not be happy with that output. The advantage of such a simple oracle is that it is very simple to implement and is a good measure of program reliability.

You can think of fuzz testing as the figurative million monkeys at the keyboard and mouse.

To put fuzz testing in context, we should note that it is just one technique out of many testing techniques. Traditionally, testing is based on checking of the program against some specification. Often, this specification is based on matching test inputs with expected outputs.

Testing techniques can be at the unit test level, applied to small parts of the code, such as to each function or method, or to whole modules or classes, or to the whole program. For complex multi-server systems – such as an e-commerce system with servers for input processing, load balancing, databases, financials, and business logic – final testing might include collectively testing of all the servers.

We can measure the effectiveness of a testing technique by how much of the code in a program gets exercised by the tests. This measure is called “code coverage”. Fuzz testing, being random, does not give you any guarantees about code coverage.

Fuzz testing is attractive because you do not have to construct an oracle based on a specification of correct behavior. However, using such a simple specification means that many incorrect behaviors may go undetected.

So, consider fuzz testing a good technique but only one tool in your testing toolbox.

38.3 Early Variations on Fuzz Testing

The most basic version of fuzz testing, illustrated in the top part of Figure 2, is using a program that generates random input streams to test a program. Your fuzz generator might have options to control the length of the stream, whether it includes zero (null) bytes and whether to include signed (8-bit) values. You can see options from the fuzz program description Figure 1.

Here, we are assuming that the program is executed from the command line and input is read from standard input or a specified file. There are a lot of programs for which this approach works well.

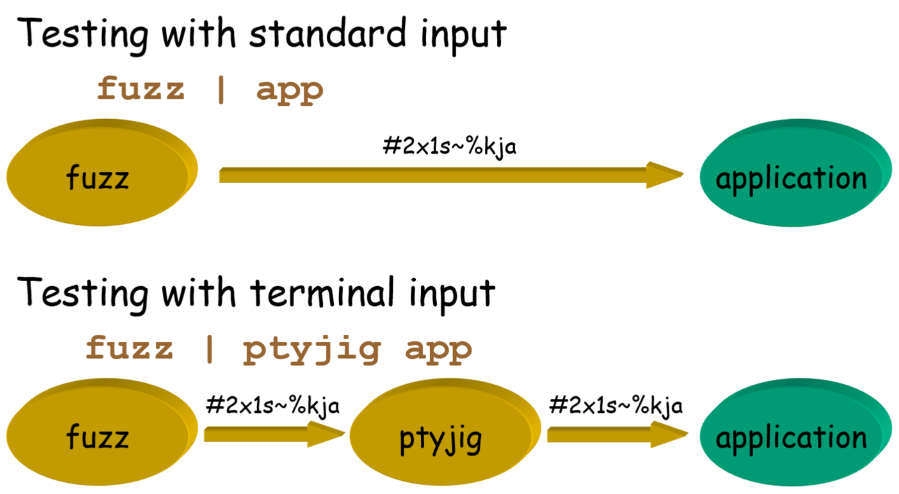


Figure 2: The Simplest Forms of Fuzz Testing: Piping Random Input to a Program

On UNIX systems, a terminal (shell) window has the ability to move the cursor around, writing anywhere on the screen. This is the functionality that supports running a text editor such as `vim` or `emacs` in the window. Programs that use this functionality need more than just text input and output; for example, they need the ability to move the cursor and not echo characters. This functionality is supported by a "pseudo terminal driver", also known as a `pty`. To test programs that use this pty functionality, we built a test jig that interfaces with it, as shown in the bottom half of Figure *2*.

A common interface to a program is a GUI, a graphical user interface based on a windowing system. Such programs use the mouse and keyboard to make selections, push buttons, use menus, fill in fields, and draw with the mouse. To test such a program, you have to be able to intercept the communication between the user input and the program and injects the test input. Such communication is based on keyboard and mouse events.

You have to build a test jig or harness to do this interception, as illustrated in Figure 3, and such a jig will be different for different systems, such as X-Windows or Wayland on UNIX, Win32 on Microsoft Windows, and Aqua on MacOS X.

The goal is to test the program with random streams of valid (or possibly invalid) keyboard and mouse events. To evaluate the results of the testing, we use the same simple oracle: did the program crash or hang?

Alternatively, you can turn this around and send random user events to the window system to see if the window system itself is reliable.

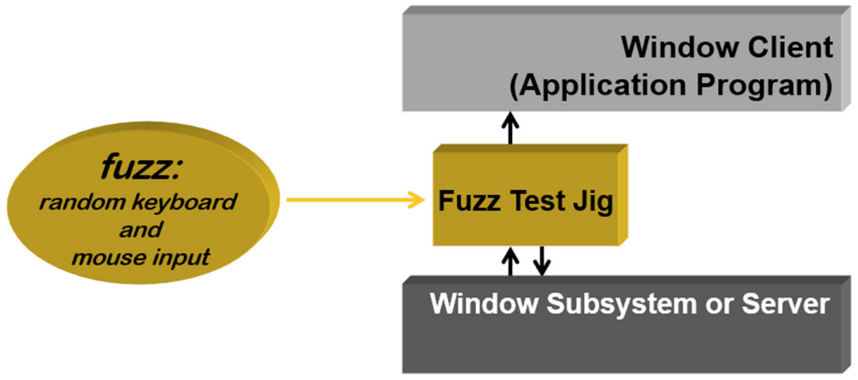


Figure 3: Fuzz Testing of GUI-based Applications

You do not have to stop at fuzz testing regular programs You can also test network services, like web servers, database servers, remote login servers, and cloud servers. These servers run on a host and listen to specific network port for new connections.

To do this type of testing, you need to build a test jig that will connect to a network service and pass random input to that service, as shown in Figure 4 for a network service.

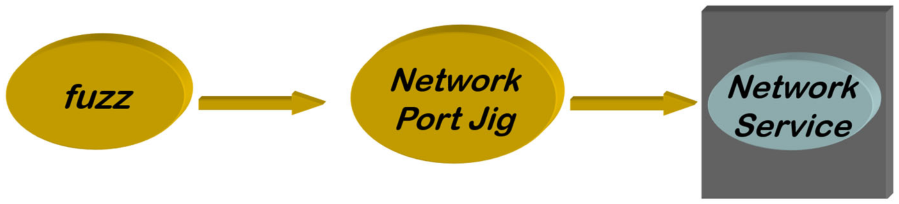


Figure 4: Fuzz Testing for Network Services

Yet another way to use fuzzing involves multithreaded programs. Multicore processors have become the norm, in your desktop computer, laptop, and even your phone. As a result, multithreaded programming is now ubiquitous and along with this type of programming comes new bugs. These synchronization bugs – race conditions and deadlocks – can be extremely hard to find.

We can expose some of these bugs by stress-testing multithreaded programs by causing the thread scheduler to choose random or extremely unusual schedules that control the order of thread execution. The test jig for this type of testing is shown in Figure 5.

Our oracle is still the same, looking for crashes or hangs to indicate failure.

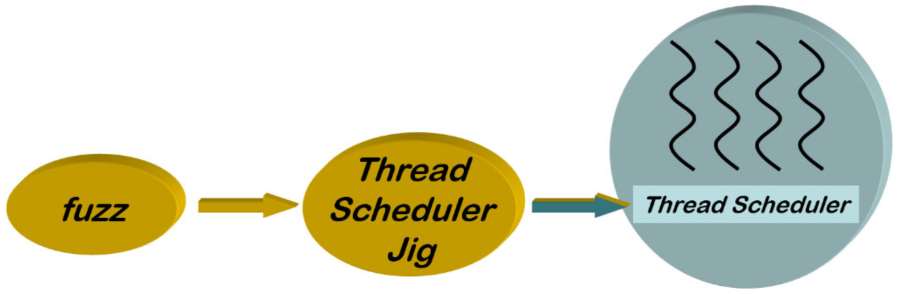


Figure 5: Fuzz Testing for Multi-Threaded Programs

38.4 The Dimensions of Modern Fuzz Testing

More recently, researchers have been pushing fuzz testing in new and important directions. As a result, we can talk about a fuzz tester (often called a “fuzzer”) by describing how it fits into three categories:

Unstructured vs. Structured: In the first case, we just generate random bytes (or other more modern character representations such as Unicode). In the second case, we know something about the structure of the input and use that structure as a vehicle to hold input with valid structure but random field contents.

Generation vs. Mutation: In the first case, the input is randomly generated each time. In the second, existing inputs are randomly modified to generate new inputs.

Black Box vs. Gray Box vs. White Box: In the first case, we know nothing about internal structure of the program that we are testing. In the second case, we know something about the structure, such as how it is divided into basic blocks. And in the third case, we know all the details of the control- and dataflow of the program being tested.

These categories graphically depicted in Figure 6. And we can see a few fuzz testing tools labeled by how they fit into these categories. As we will see from this section, the classic fuzz testing would now be described as black box, unstructured, and generational.

The following sections look at each of these categories in more detail.

38.4.1 Unstructured vs. Structured

The first category we examine is structured vs. unstructured input. Unstructured input is just a random stream of byte values (or other character representation). Structured input tries to take into account the input syntax and vary the contents of the information in the input fields within the valid syntax. Of course, unstructured input is the simplest type to generate.

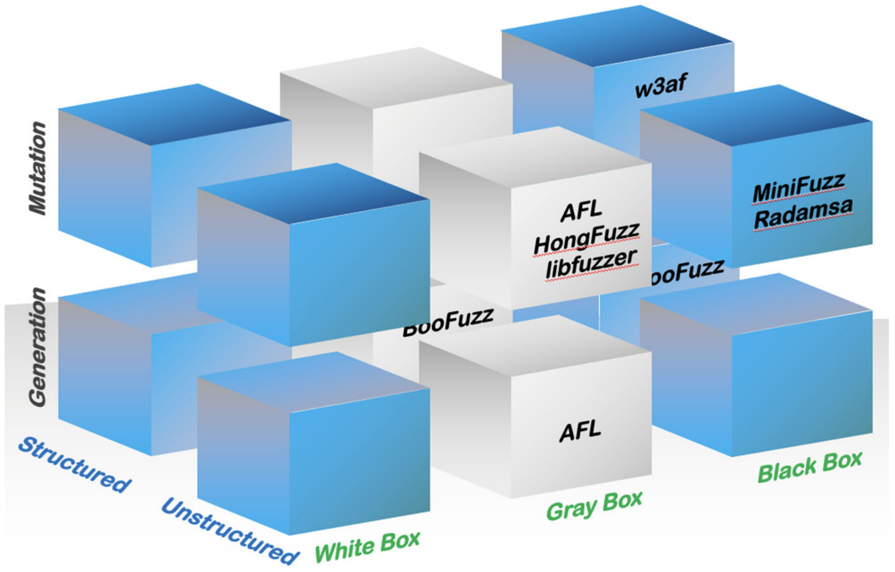


Figure 6: The Three Dimensions of Modern Fuzz Testers

The motivation to use structure input is to try to test more deeply into a program's logic.

In Figure 7, we show a common program structure. Input comes into the program and is first checked for valid format. The input is then processed to understand what kind of operation needs to be done. Finally, the individual operations are done deep in the program's logic.

If we use unstructured input, many of the bugs that we are likely to find will be found in the input-checking stage. Since the input bytes are totally random, they are unlikely to be valid operations, so will often be rejected at this early stage (or cause a crash at this stage).

Of course, some inputs may make it deeper into the program's logic, but perhaps this is not likely enough to thoroughly test the program's code.

So, we can partially structure the input, maybe starting each line with a valid keyword and including valid operators in the random bytes that follow:

```
select x1s~%kjaF.=|2qa
```

This will take us deeper into the program's logic and (click) expose more bugs at this level. We can further structure the input to try to test deep into the program's structure. We might use completely valid syntax or field structure for the input, with random contents for each field:

```
select from ,$&234k where x = !~%jF=2a
```

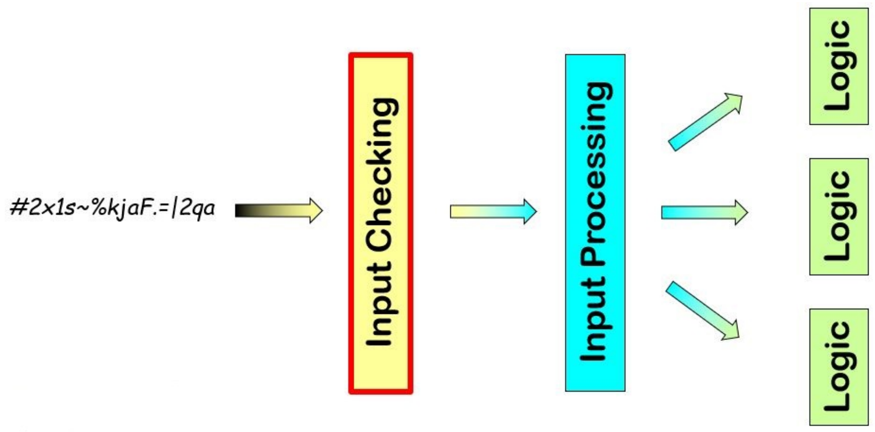


Figure 7: Conceptual Layers of a Program's Structure

If we successfully create such structured random input, we might be able to expose bugs at a deeper level.

Generating structured input is definitely more work than generating random byte strings. For each tested program, you have to be able to specify what is valid syntax and then generate input streams that conform to that syntax. So, what can we accomplish by structuring input? For programs that process command lines, structuring input will get past the basic input syntax processing and into the logic of the commands.

For window systems, purely random input maybe not generate valid keyboard and mouse events. For example, valid keyboard input requires that each key-down event is matched with a corresponding key-up event. And valid mouse events must have X-Y coordinates within the size of the window. Again, structuring the input to be valid format allows testing past the input-checking stage.

Network protocols are similar to command line programs in that they have a specific syntax. Conforming to that syntax allows you to test more of the server's logic.

Compilers are extremely complex programs and difficult to test. So randomness offers a chance to explore parts of the compiler's logic that you might not think about. However, if the syntax of the test input isn't valid, then you might leave large parts of the compiler logic untested.

38.4.2 Generation vs. Mutation

A second category for fuzz testing is generation vs. mutation testing.

The first fuzz tester simply generated new random inputs for each test. Later, researchers developed techniques that would start with an existing input, often a valid one, and then mutate it – i.e., modify it in some random way – to generate the next test input.

For example, you could start with a valid SQL query and modify parts of it to test the database system. Or start with a valid Java program and modify parts of it to test a compiler. Or you could simply randomly or systematically modify a previous test input to generate a new one to try to explore more of the program's state space resulting in getting better code coverage.

Mutation testers have been widely used in recent years, and in Chapter 42, we will learn about the AFL fuzz tester that uses this technique.

38.4.3 Black Box vs. Gray Box vs. White Box.

The last category of fuzz testing is based on how much do you know about the program that you are testing?

The first fuzz tester used the simplest form of testing, black box testing. This means that you know nothing about the structure of the program; you just feed it inputs and see whether it crashes or hangs.

However, if we can know something about the program that is being tested, for example, what parts of the code were tested, we could understand how better to generate new inputs and evaluate the meaning of the test results. So, a gray box tester tracks which parts of the program you have tested. This means that you need to instrument the code, by modifying the source or binary, to track which parts of the program were executed. The tester still does not understand the functionality of the program but does track what parts got executed.

If you analyze the program structure and functionality, you can track how the program executed. For example, you can track the `if` statements in the program to see if a given input followed the true or false path, and then modify the input to try to explore the other path. This kind of testing is called white box testing. It is the most powerful of the three approaches, but also the most difficult to implement. While some research projects have tried this approach[2], no fuzz tester in common use is based on white box testing.

[2] P. Godefroid, M.Y. Levin and D. Molnar, "Automated Whitebox Fuzz Testing", *16th Annual Network & Distributed System Security Symposium (NDSS),* San Diego, California, February 2008.

38.4.4 Mutational Gray Box Testing (Coverage Guided Testing)

If we design a fuzz tester that used mutational and gray box testing, we end up with an important class of contemporary fuzz testing called *coverage guided testing*. Tools such as AFL (which we will learn more about in Chapter 42), libfuzzer and HongFuzz are all cover guided testers.

We can see the structure and workflow of a coverage guided fuzz tester in Figure 8. These testers start with a set of inputs, called “seeds”, provided by the programmer or analyst. The fuzz tester chooses (“pick”) one of these seeds and then mutates it in some way (“mutate”) to generate the test input.

The tester then runs the program using this mutated input (“execute”) and hopefully generates a crash or hang, recording feedback information about what input was used and what part of the program executed (“Execution Feedback”). Since the tool tracks and records which blocks execute, it is a gray box tester.

The tester then uses this feedback to decide if this execution of the program is “interesting” in that it tested new parts of the program that had not been tested before. The recording of blocks execute allows the tools to decide it new parts of the program have been tested.

If the result was interesting, then the input used is saved as part of the collection of seeds, and the cycle starts over again. Such a cycle continues until the programmer runs out of time or finds enough bugs to keep them busy.

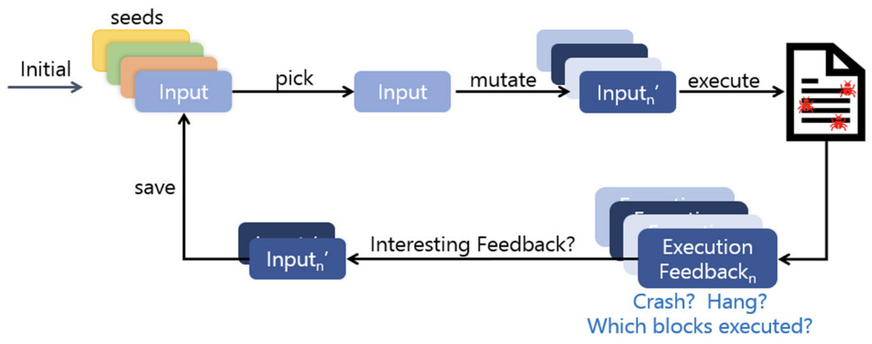


Figure 8: The Structure and Work Flow of a Coverage Guided Fuzz Tester

38.5 The Path Through History

Fuzz testing has become an enormous area of research and practice. It has gone far beyond the simple tool developed in the University of Wisconsin-Madison in 1988. A quick check on Google Scholar for publications with “fuzz testing” in their title or abstract produced more than 76,000 results.

And there is wide adoption in the software industry. For example:

Google provides OSS-Fuzz, a free open source fuzz testing platform[3].

Microsoft provides Project OneFuzz[4], a Fuzzing-as-a-Service (FaaS) platform originally used internally by Microsoft and now open to the public.

Amazon provides a fuzz testing service[5] as part of AWS Device Farm for testing Android and iOS applications.

The initial fuzz testing work at UW-Madison has spanned over 30 years, with the latest study only a few years ago. Note that **all** of these studies are based on class projects in our graduate operating system class. The main results from these studies includes:

1990[6]: The initial fuzz testing: it was easy to crash lots of UNIX utilities and application programs.

1995[7]: More UNIX applications and utilities were tested on more UNIX platforms. Plus X-window GUI-based applications and network services were test. Even though the fuzz results had be published and freely shared, crashes were far too easy. Note that this is the first time that open source systems were tested and they fared better than the commercial ones.

2000[8]: It was time to test Windows GUI-based applications. It was even easier to crash these GUI applications than in the previous study.

---

[3] `https://bughunters.google.com/open-source-security/oss-fuzz`

[4] `https://www.microsoft.com/en-us/research/project/project-onefuzz/`

[5] `https://docs.aws.amazon.com/devicefarm/latest/developerguide/test-types-built-in-fuzz.html`

[6] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM* **33**, 12 (December 1990). Also appears (in German translation) as "Fatale Fehlertractigkeit: Eine Empirische Studie zur Zuverlassigkeit von UNIX-Utilities", *iX*, March 1991.

[7] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", *Computer Sciences Technical Report #1268*, University of Wisconsin-Madison, April 1995. Appears (in German translation) as "Empirische Studie zur Zuverlasskeit von UNIX-Utilities: Nichts dazu Gelernt", *iX*, September 1995.

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

2006[9]: We had not yet tested everyone's favorite platform, the Apple Mac. While the command line applications did pretty well (though not perfect, by any means), GUI-based applications on Mac OS X were even worse than previous GUI-based studies.

2020[10]: After so many new innovations happened in fuzz testing, we wanted to see if the classic (and simple) tools were still useful and relevant. Sadly, these classic tools still found too many bugs. Many of these bugs were the same type as found in previous studies.

Missing from the body of this work are complete longitudinal studies of each system, where in each study, each system was tested. For example, we know that the GUI results got steadily worse starting from the 1995 UNIX X-Windows study to the 2000 Windows study to the 2006 MacOS study. This leads to the unanswered question: was the increase in failure rate due to the progression from UNIX to Windows to MacOS or due to how software changed over that time period or some combination of these?

In the next four chapters, we will discuss these results and others:

- Command line studies at the UW-Madison, including results from the 1990, 1995, 2006, and 2020 publications.
- GUI (Window) studies from UW-Madison from 1995, 2000, and 2006.
- Other studies UW-Madison, including window servers, network services, and return value checking.
- An introduction to a popular and widely used coverage guided fuzz tester, AFL (American Fuzzy Lop).

38.6 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, also known as gray box mutation testing

[9] B.P. Miller, G. Cooksey and F. Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing", *First International Workshop on Random Testing*, Portland, Maine, July 2006.

[10] B.P. Miller, M. Zhang and E.R. Heymann, "The Relevance of Classic Fuzz Testing: Have We Solved This One?", *IEEE Transactions on Software Engineering* **48**, 6, June 2022.

38.7 Exercises

1. In Section 38.4.1, we presented several applications of structured fuzz testing. Think of new types of applications to apply structured input data for fuzz testing. 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 which parts of the code were tested in this application? Would structuring the random input all you to test deeper into the call graph of the program?
2. 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?