Rabbit in the Loop
A primer on feedback directed fuzzing using American Fuzzy Lop
by Kevin Läufer <laeuger@cs.berkeley.edu>

Abstract

This guide aims to provide the reader with a good intuition about the process of feedback directed fuzz testing as well as some vocabulary that makes it easier to communicate about this area of research.

Introduction

It is surprisingly difficult to write programs that do not contain bugs. Some might even argue that it is impossible (at least in the general case). Experience over the last decades has shown that parsers and deserializers written in non-memory safe languages like C and C++ are especially susceptible to security critical bugs.

When some of these programs, like readelf were designed, the internet was in its infancy and the expected use-case was for the end user to run readelf on trusted binaries on their computer. The same was true for png parsers like libpng. With the advent of the Web 2.0, it seems like every library or command line tool will eventually be used in a web service and thus process untrusted inputs. But even parsers that we originally intended to be used on the internet, such as url parsers have a surprising amount of bugs. Since many of these programs are written in non-memory-safe languages (mostly because the author was familiar with these languages or for speed and compatibility), a lot of bugs turn out to be exploitable for remote code execution.

Already in 1990, Miller at.al performed a study of the reliability of UNIX utilities and found that they were able to crash more than 25% by simple random input generation [0]. More recently, feedback directed fuzzing has emerged as a much more powerful technique that has been used to find security vulnerabilities in a large range of widely used applications. One powerful and yet easy to use tool is American Fuzzy Lop (AFL) which was created by Google researcher Michal Zalewski and can be downloaded on the following website which also contains a trophy case of important bugs that were discovered using AFL: http://lcamtuf.coredump.cx/afl/

This guide aims to provide the reader with a good intuition about the process of feedback directed fuzz testing as well as some vocabulary that makes it easier to communicate about this area of research.
An Intuitive Model of Fuzzing

“Intuitive” vs (Rigorous) “Mathematical” Model

In order to wrap our head around a new subject it helps to look at a simplified model of what is going on. This model won’t be an exact description of reality but sometimes a little lie-to-children helps us get started in appreciating a new topic. We want to present an “intuitive” model instead of a rigorous mathematical one as proposed by the authors in [1]. We believe that this model will be easier to understand as it does without convoluted mathematical notation, while still providing the same benefits. When reading [1] carefully it becomes clear that the major innovations proposed stem from the intuition built through the model and are not rigorously derived from the mathematical formulation. Thus it should not be a loss to never include them. Instead we try to highlight most of the simplifications we make such that the reader has a good starting point to learn more about what is really going on.

Program Under Test, Input Space and Random Testing

[DEFINITION] Program Under Test: First, let us talk about the program that will be fuzzed. We assume that it takes a single file as input which is parsed and processed to produce a result. Most programs that are tested with fuzzers take less than a millisecond to process a single input and thus we can test at least about 1000 different inputs per second. The program under test is assumed to be deterministic, which in this context means that when we run it with the same input we will get the exact same execution. This requirement can be relaxed in practise, but this is outside of our model.

[DEFINITION] Test Input: Each test input consists of a number of bytes. It is reasonable to restrict the maximum size of a test input since our processing resources are finite. A lot of inputs are around 1kiB, most are under 1 MiB. Since we assume the program under test to be deterministic, an input can be mapped to exactly one execution of the program.

Now, let us assume that the input size is limited to be a maximum of M bytes. Under this constraint the maximum number of possible inputs is:

\[
\left(2^8\right)^0 \cdot \left(2^8\right)^1 \cdot \left(2^8\right)^2 \cdot \ldots \cdot \left(2^8\right)^M = \left(2^8\right)^{0+1+2+\ldots+M} = \left(2^8\right)^{\frac{M(M+1)}{2}} = 2^{4M(M+1)}
\]

This input space is huge and accurately modelling the distance to precisely reflect the hamming distance (the number of bit flips) between two inputs would require a high dimensional space which would be impossible to visualize\(^1\). Thus we will model a small slice of the input space as a 2D grid of disks, each representing a unique input, in which the distance between two disks reflects how different two inputs are:

\(^1\) At least the author of this guide lacks the imagination and drawing skills to do so.
Figure 1: input space as a set of round cards, bug that we want to discover marked in red

We can view this input space as a card game in which all cards lay face down on the table. We are only allowed to flip one card at a time and it takes some time T to do so (i.e. the time it takes to run our program under test on the input). Once we have run the input (once we turned over the card) we can see whether we uncovered a bug (i.e., if the program crashes). The problem with this game is the astronomically high number of cards, with only a small number of them revealing a bug once turned over.

In Figure 1, we marked the input that will reveal a bug in our program in red. Keep in mind, that the fuzzier, which is the player in our card flipping game, does not have this information. It is in fact the goal of the fuzzing process to eventually try out the bug causing input (flip the correct card) and thus discover a new bug in our program under test.

In random fuzzing as used in [0], we would just randomly chose cards to flip, hoping that one of them will reveal a bug.

Sampling the Input Space Through Mutations

A different approach is to start with a seed input and to explore the input space from there. The seed can be empty, but it is normally much more effective to choose a valid input file as a starting point. The seed input is mutated by the fuzzier to produce new inputs, thus sampling from the space around the seed. This approach is illustrated in Figure 2 and 3.
Figure 2: AFL starts with a seed (green) and tries to explore other inputs through mutation.

Figure 3: mutations (arrows) samples the input space around the seed (green), but may never reach the bug (red).

Mutation strategies can be split into **deterministic** approaches which will generate the same results every time they are run on the same seed and **non-deterministic** strategies which sample from a probability distribution when picking the location in the seed to mutate and the
exact mutation to use. **Table 1** contains an overview of all deterministic mutation strategies used in AFL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitflip 1/1</td>
<td>flip single bit</td>
</tr>
<tr>
<td>bitflip 2/1</td>
<td>flip two adjacent bits</td>
</tr>
<tr>
<td>bitflip 4/1</td>
<td>flip four adjacent bits</td>
</tr>
<tr>
<td>bitflip 8/8</td>
<td>flip single byte</td>
</tr>
<tr>
<td>bitflip 16/8</td>
<td>flip two adjacent bytes</td>
</tr>
<tr>
<td>bitflip 32/8</td>
<td>flip four adjacent bytes</td>
</tr>
<tr>
<td>arith 8/8</td>
<td>treat single byte as 8-bit integer, add/sub values from 0 to 35</td>
</tr>
<tr>
<td>arith 16/8</td>
<td>treat two adjacent bytes as 16-bit big/little endian integer, add/sub values from 0 to 35</td>
</tr>
<tr>
<td>arith 32/8</td>
<td>treat four adjacent bytes as 32-bit big/little endian integer, add/sub values from 0 to 35</td>
</tr>
<tr>
<td>interest 8/8</td>
<td>replace single byte with <em>interesting</em> 8-bit integer values, such as: -128, -1, 0, 127, 100 etc.</td>
</tr>
<tr>
<td>interest 16/8</td>
<td>replace two adjacent bytes with <em>interesting</em> 16-bit integers values, such as: -32768, 255, 512, 1024, 32767</td>
</tr>
<tr>
<td>interest 32/8</td>
<td>replace two adjacent bytes with <em>interesting</em> 32-bit integers values, such as: -2147483648, -100663046, 2147483647</td>
</tr>
<tr>
<td>user extras (over)</td>
<td>overwrite bytes with interesting tokens</td>
</tr>
<tr>
<td>user extras (insert)</td>
<td>insert interesting tokens</td>
</tr>
<tr>
<td>auto extras (over)</td>
<td>overwrite bytes with learned interesting tokens</td>
</tr>
</tbody>
</table>

**Table 1: Deterministic** Mutation Strategies used in AFL

The **non-deterministic** strategy in AFL is called the *havoc* stage and mostly consists of applying a random set of up to 128 different sub strategies to random locations of the input. The author of this guide is not aware of any good analysis of this strategy. However, experience has shown that quite a lot of progress is made in this non-deterministic stage. Some of the results in [1] can apparently be explained by AFL spending more time on non-deterministic mutation [2].
Feedback Directed Fuzzing

While we laid out in the previous section how mutation can be used to sample from a part of the input space we were awfully quiet about how this helps us find bugs better than random fuzzing. The reason for this is that - at least in general - it does not (much)².

![Figure 4](image-url)

**Figure 4**: we introduce intermediate coverage goals that signal our fuzzer that it has made progress; the yellow disks represent inputs that - if executed - will archive new coverage.

The ingredient that we are missing is feedback. If our fuzzer is able to tell if it has made progress towards its goal of finding bugs, it should remember the input so that it can be used as seed for further mutations. **Figure 4** shows the inputs which - when executed - will reveal that some progress was made. **Figure 5** and **6** show how the feedback helps us pick new inputs to mutate which (might) bring us closer to our goal.

Next we need to discuss how to define and measure progress in this context. Unfortunately we cannot know exactly how far away from discovering a bug our fuzzer is (unless you already know where the bug is³). Intuitively we want the fuzzer to cover all possible executions of our program. To make this more precise we need to get familiar with some definitions from the compiler and program testing literature.

² Starting from, e.g. a valid PDF and mutating it might give you better results than sending random bytes to a PDF parser. However random mutations can also be seen as a particular implementation of random fuzzing.

³ But if you know where the bug is, why are you fuzzing in the first place? You must be a researcher in software testing trying to demonstrate that their tool works by finding bugs that people already know and no longer care about. I am sorry.
Figure 5: when running the mutated seed input, the fuzzer discovers that one of them makes progress (yellow with green border)

Figure 6: the fuzzer explores the input space by mutating the newly discovered input and thus finds the bug causing input
[DEFINITION] Execution Equivalence: Let us talk a bit more about what it means for a program execution to be the same. For this discussion we will ignore context switching, out-of-order execution as well as modern memory management mechanisms and security features such as address space layout randomization (ASLR). In this context for a program execution to be the same would mean that the same sequence of instructions is executed on the same data.

[DEFINITION] Control Flow Graph: The above definition is not very useful in practise and thus needs to be relaxed. In compiler research, programs are divided into basic blocks that contain atomic sequences of instructions and edges that describe all possible successor basic blocks, thus forming the control flow graph. An if statement in a C program introduces two edges: one in case the condition is false, one in case the condition is true. If we only had if and switch statements, the control flow graph would be a directed acyclic graph (DAG). Realistic programs, however, contain loops or recursion and thus cycles are introduced into the graph.

[DEFINITION] Program Path: A program path is a path through the control flow graph. It is obvious that with the introduction of cycles, there is essentially an infinite amount of such paths. When run with a fixed input, the program under test will always take the same path. Thus test run, test input and program path are often used interchangeably. However, please note that the same program path might be triggered by a set of different inputs. Finding the smallest input that covers (more precisely: that makes the program cover) a given path is one example of test input minimization.

[DEFINITION] Branch Counts: While the program path metric ignores the exact input data, it is a very precise representation of the control flow. This can be undesirable. As an example, take an if statement in a parser that checks the first bit and writes a value to a memory location. This functionality might be completely unrelated to the further execution of the program. However, this single if statement will double the number of possible paths. With fuzzing we need to balance the need for precision in order to explore different functionalities in programs, with the resource constraints in order to find bugs in a reasonable amount of time.

AFL uses branch counts as a metric to measure progress. Through clever engineering, new feedback is generated whenever the program under test is run with a new input. This is possible by using a LLVM compiler pass to find all statements that cross from one basic block to another (i.e., branch statements) and to then add a small amount of custom code. This code computes a simple hash of ids that identify the source and the target basic block and thus the transition. It then uses the computed value to index into an array in shared memory and increment an 8-bit counter at this location. In order to avoid counting every iteration of a loop as interesting, the counts are sorted into exponentially expanding bins. Thus going from one count to two is considered interesting progress, while going from 7 to 8 is not.

Other AFL Features
AFL has a range of features that we did not talk about in this report. While it saves all interesting inputs that it finds, it has a mechanism to pick amongst these inputs to decide which one it wants to mutate next. There are also a lot of implementation details we glossed over (such as the probabilistic nature of AFL’s branch counter map) and probably also some important features that the author of this guide is not aware of.
Running AFL

Having talked about the intuition and the basic mechanisms behind feedback directed fuzzing with AFL, we want to invite the reader to try out AFL and see for themselves how it works. We strongly recommend using a Linux based system to do so as AFL has some serious performance issues on MacOS. A recent version of the Clang compiler makes things easier as it allows us to use the LLVM instrumentation mode instead of the legacy version which operates directly on the assembly of the program.

You can download the latest version of AFL from its website: http://lcamtuf.coredump.cx/afl/

The website as well as the sources also contain the excellent Quick Start Guide which we won’t repeat here: http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt
To use the LLVM mode, change to the llvm_mode directory and run make. This will generate afl-clang-fast which can be used to replace afl-gcc.

An easy program for initial experiments is the minigzip test program from the zlib: http://zlib.net/

Expect to run the fuzzer for about 1 hour for initial experiments. Most papers eventually evaluate their technique on runs that take between 6 and 24 hours. Try out different seeds in the input directory: Try to see how using a valid zip file as starting point changes the outcome of the fuzzing run compared to using an empty file.

In case you want to use custom feedback to guide the fuzzer for a particular program. The author of this guide has a C library available to do so: https://github.com/ekiwi/custom-afl-instrumentation
You could try out different feedback techniques to help the fuzzer solve a maze.

We wish you Happy Fuzzing! and hope that this guide was useful.
Feel free to get in touch via email if you have questions or suggestions: laeufer@cs.berkeley.edu

References

[0]: Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM. 1990 Dec


---

4 The problem is the fork system call which is used by AFL in a creative way to restart the program under test as quickly as possible: https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html
5 This is most important if one wants to modify the instrumentation as the implementation using LLVM is much easier to work with.
6 There is some debate in the community on how long to run the fuzzer during the evaluation.
[2]: Mailing List conversation between the AFL author and Marcel Böhme:
https://groups.google.com/d/msg/afl-users/1PmKJC-EKZ0/zck6lu77DgAJ (retrieved on December 15th 2017)