Chapter 2

Background

2.1 Literature Review

Sutton et al. [1] defined the procedure of fuzz testing, as shown in Figure 2.1:

1. Generally speaking, the first stage is to identify the target. A target may be a software or a combination of both software and hardware [2]. The targeted software may be the software itself, or any part of the software, for instance, a library or an API. In the rest of this article, our target is software.

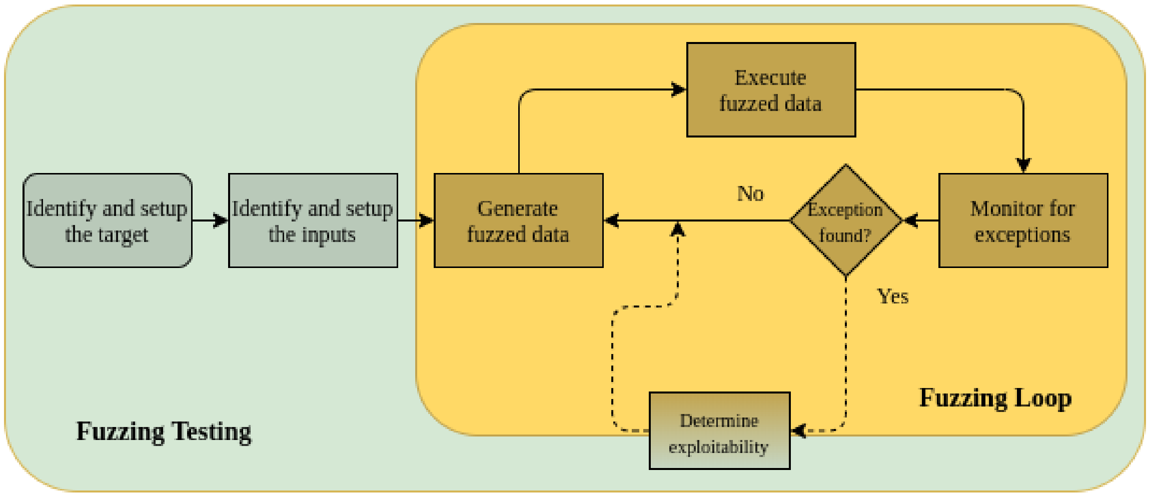

Figure 2.1: Fuzzing phases

2. To execute the target program, we need to specify how the program parses the inputs. For instance, the target program may take several command-line arguments as the input, or the arguments are file locations for reading data. Environmental variables, file formats, and any other variables that may change the program's execution are all considered an input, and a fuzzer can choose them for the execution of the target program. In any case, the fuzzer needs to know the data that would cause a vulnerability. The fuzzer needs a set of seeds for initialization. This set could be empty, and a fuzzer without any sample inputs may find some valid files out of thin air [3] that are acceptable by the target program.

3. The generation of fuzzed data is the beginning of the fuzz testing loop. The target program needs a queue of inputs for testing the execution. New inputs are generated either by a dumb mutation of the content of input or generation of new content for input intelligently. The prior is called mutation-based fuzzing, and the later is generation-based fuzzing. Our fuzzer is mutation-based. In chapter 3, we will discuss more how a mutational dumb fuzzer can work intelligently.

4. A fuzzer takes the processed inputs and executes the target program using the provided inputs. Depending on the resources needed for the execution of the target program, this stage can be the bottleneck for fuzzing. Generally, the output of this stage is any detected misbehavior during the execution over an input.

5. If an exceptional event happens during the execution of the program that prevents the program from exiting successfully, we say an exception has occurred. These exceptions are the vulnerabilities that may be exploitable and may cause security problems. Handling an exception properly and pinpointing the input

(or probably part of the content of the input) responsible for the exception is the primary purpose of this stage of fuzz testing.

6. A vulnerability may be harmless! Suppose we find an exception (vulnerability) in a module not handled within the program, and the operating system catches it. If another program uses this module, and the program is first validating the input for the module, then the domain of inputs for this module is constrained and filtered by the program, and the vulnerability may never occur. If we can find an exploitable vulnerability, the fuzzer must detect it and report it as an exploitable vulnerability. The generation phase may need this report for later usage.

The introduced fuzzer by Miller [4] was of the very first naive blackbox fuzzer targeting a collection of Unix utilities, on different Unix versions. It runs the fuzzing for different lengths of inputs for each utility (of the total 88 utilities) and expects a **crash**, **hang**, or a **succeed** for the execution of the program. The generation of inputs is by mutation of contents of inputs, and the target program is a blackbox. As a result, this fuzzer is a mutation-based blackbox fuzzer.

A blackbox fuzzer randomly generates a different stream of inputs and does not decide how to choose those generations to enhance the performance of the fuzzer. Another downside of this fuzzer (and other blackbox fuzzers) is when the program faces some branches with magic values, constraining the variable to a specific set of values; the fuzzer has to apply the exact values for the constraining statement which may have a very low probability - a 4 bytes word is created with $1/2^{32}$ probability. Nevertheless, when the target application is known, the performance of the blackbox fuzzing technique can be increased drastically. In [5] and [6] some network protocols are fuzzed as blackbox. Any application on the web may be considered as a blackbox program as well, so as [7] and [8] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

SAGE, a whitebox fuzzer was developed as an alternative to blackbox fuzzing, to cover the lacks of blackbox fuzzers [9]. A fuzzer is a whitebox fuzzer if we have the source code of the target program, and the fuzzer can use the information within the source code. With the benefit of having the source code and internal knowledge for fuzzing, a whitebox fuzzer can leverage symbolic constraints for symbolic analysis to solve the constraints in the program [10]. It can also use dynamic, and concolic execution [11] and use taint analysis to locate the regions of seed files influencing values used by program [12]; whitebox fuzzing can also find the grammar for parsing the input files without any prior knowledge [13]. These techniques help in finding code coverage and a more intelligent search for vulnerabilities.

Greybox fuzzing resides between whitebox and blackbox fuzzing, as we only have partial knowledge about the internals of the target program. We do not have the source code, but we have some knowledge about the program (for instance, we have the binary file), and as a result, we have the instructions for the program, which can be helpful for the fuzzing. A handy feature for an execution is the code coverage of the run. A coverage-based greybox fuzzer keeps track of the different inputs with different coverages and may generate inputs for the next generations based on the new code coverages found in each iteration. To detect the coverage, we need an instrumentation [14] that we will explain more.

Coverage-based testing benefits from instrumentation to discover which parts of the (binary) code is covered. Some criteria, such as data coverage, statement coverage, block coverage, decision coverage, and path coverage, can be used to investigate the overall code coverage. [15] Bohme et al. [16] introduced a coverage-based greybox fuzzer that extends AFL and benefits from the Markov Chain model. The fuzzer looks for the paths that were covered using seeds, and when a new path is seen, it is stored accordingly. Then it reports a probability that after fuzzing an input with path i, it generates an input with path j. The fuzzer chooses a new collection of seeds to fuzz according to the energies (inversely proportional to the density of the stationary distribution) assigned to each seed. The fuzzer generates more inputs from an input with higher energy. In another article by Bohme et al. [17], they check the code-coverage to provide inputs that guide the program execution toward some target locations. Some of the applications of such fuzzing techniques are patch testing and crash reproduction, which has different uses compared to other coverage-based fuzzers.

The types of vulnerabilities that a fuzzer is involved with may be different from other fuzzers. For example, AFL looks for crashes or hangs by selecting and mutating the inputs, and at the same time, considers the code coverage, size of the input, and the execution time of the target program. PerfFuzz [18] is another greybox fuzzer based on AFL, which aims to generate inputs for executions with higher execution time, while using most of the features of AFL in exploring for more coverage. PerfFuzz counts how many times each of the edges of the control flow graph (CFG) is executed. Using SlowFuzz [19], we can consider any type of resource as a feature to detect the worst-case scenarios (inputs) for a given program. In another project based on AFL, Memlock [20] investigates the memory exhaustion by calculating the maximum runtime memory required during executions.