Human-Controlled Fuzzing With AFL

Maxim Grishin
Bachelor of Information Security, MEPhI; Moscow, Russia

Igor Korkin, PhD
Security Researcher; Moscow, Russia

HUMAN-CONTROLLED FUZZING WITH AFL

Maxim Grishin
Bachelor of Information Security, MEPhI
Moscow, Russia
grisinmaksim096@gmail.com

Igor Korkin, PhD
Security Researcher
Moscow, Russia
igor.korkin@gmail.com

ABSTRACT

Fuzzing techniques are applied to reveal different types of bugs and vulnerabilities. American Fuzzy Lop (AFL) is a free most popular software fuzzer used by many other fuzzing frameworks. AFL supports autonomous mode of operation that uses the previous step output into the next step, as a result fuzzer spends a lot of time analyzing minor code sections. By making fuzzing process more focused and human controlled security expert can save time and find more bugs in less time. We designed a new module that can fuzz only the specified functions. As a result, the chosen ones will be inspected more meticulously by a fuzzer, without wasting the time on inspecting minor code sections. The module provides API so that an expert can change which code functions need work in runtime. The module has been integrated with AFL and successfully responds to the challenge.

Keywords: software security, dynamic analysis, fuzzing, AFL.

1. INTRODUCTION

Fuzzing is a popular method of dynamic program analysis. It is a technique of automated testing when a program receives specially modified, incorrect data that can lead to its emergency state or undefined behavior. Of course, with the help of fuzzing, it is possible to identify a large number of errors and at least a large number of vulnerabilities that can lead it to incorrect behavior.

Fuzzer uses input data to modify them using mutation and generation algorithms. It repeatedly passes them to the input of the tested program. As usual, the data is changed in such a way as to increase the coverage of the basic blocks of the program code.Here are the main stages of a general fuzzing process:

- To determine the purpose of fuzzing.
- To determine the protocol and input data type.
- Changing input data using mutation and generation algorithms.
- Program execution with modified data.
- Error detection based on coverage metrics.

2. AFL TOOL FEATURES

This section covers the analysis of the AFL fuzzing features.

2.1. SCHEME AND MODES

The following steps describe the principle of AFL, see Figure 1:

1. Code instrumentation.
2. Moving data to a queue.
3. The next input is extracted from the queue and trimmed to the smallest size, which does not change the behavior of the program.
4. The input is mutated using mutation algorithms.
5. The program receives a mutated input.
6. If the input has led to a new state of the program, this input is added to the queue.
7. Go to the step 1.

AFL has built-in modes (Zalewski, 2020) like Syzygy and **QEMU**. The first mode allows you to work in the tool *instrument.exe*. **QEMU** is a mode that allows AFL to realize software fuzzing without a source code file. The binary file instrumentation has been added to the *qemu tcg* binary translation engine. AFL has a build script *~/AFL/qemu_mode/build_qemu_support.sh*, the result of which is an *afl-qemu-trace* file that emulates working in *afl-qemu* mode.

Master and **slave** modes allow you to run parallel fuzzing processes as multiple instances on multiple cores. The threads are regularly synchronized and exchange data about the found paths. It is good practice to run multiple threads, since threads in the

slave mode choose the mutation algorithm randomly, but in the master mode, one type of mutation is repeatedly applied. This can explain why slave instances find bugs faster and in greater numbers compared to master ones.

2.2. ANALYSIS OF FUZZING WITH AFL

Despite the considerable list of advantages, AFL has some disadvantages, see Table 1. It is a single-platform tool, that means that AFL is intended to be deployed on UNIX systems. The method of instrumentation based on random number generation at a certain program size (many basic blocks) increases the probability of collisions, which can lead to a situation where two different tuples will have the same numerical value and a certain part of the unique traces will not be recorded.

AFL can also spend a lot of time implementing some steps in the fuzzing process. For example, it can be the selection of an input for an offset relative to an **if-block**. Or the selection of a mutation algorithm to obtain a new state of the program. These two problems exist at different levels: the first is at the level of basic blocks, and the second one is at the level of execution traces.

In the first case, there is a solution that allows you to determine a branch that is incident to an **if-block**, with a large number of blocks not yet visited. This is the so-called "unidirectional branches" method. In this case, the use of a debugger and a recursive search algorithm is required.

In the second case, it is possible to collect information about the number of traces on a certain set and information about the number of new tuples that were obtained during the mutation of the data of this set. We can update the information periodically for each element of the queue. If an element with the "best statistics" is found, redirect the execution flow: change the order of the queue elements and the pointer of the current element.

The software implementation of the second solution is proposed in this paper. As you can see, this is an easy way to increase code coverage and, as a result, find more traces (including emergency ones) for a certain period without resorting to recursion.

There is an approach called "directed fuzzing". It is based on the fact that static analysis is first applied to determine the blocks that may contain a vulnerability, and then the maximum subgraph (a connected graph with a maximum subset of such blocks) is dynamically investigated using fuzzing. In relation to AFL, this approach has not been implemented, although its independent implementation as a fuzzer during tests had better results than AFL.

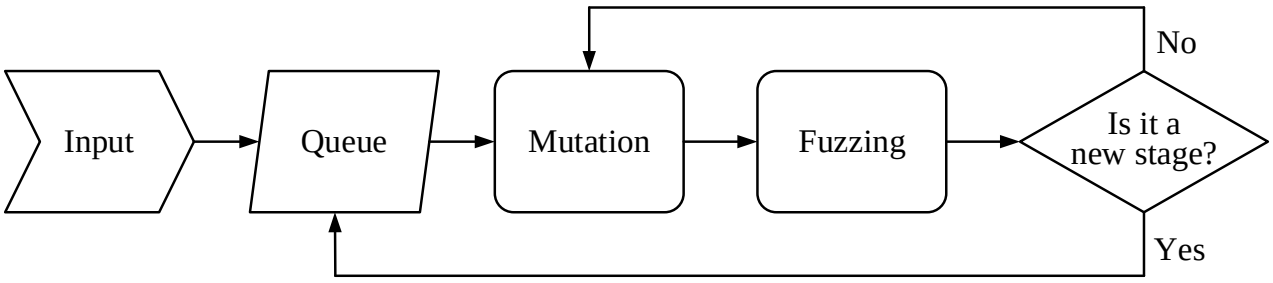


Figure 1 AFL algorithm scheme

Table 1. Analysis of AFL fuzzing.

Feature	Technical Problem	The possible third-party solution
Single-platform	unsupported	WinAFL usage
External mutators	unsupported	–
Multithreading	supported	–
Interaction with sanitizers	supported	–
QEMU - mode	supported	–
Multithreading	supported	–
Finding such error types as *memory_leak* and *out_of_memory*	unsupported	integration with libFuzzer

3. PROGRAM TOOL DEVELOPING FOR INTEGRATION WITH AFL

The problem is analyzed and a list of functional requirements for the program module (PM) is formed in this section. As already noted in the previous section, AFL in the fuzzing process spends a lot of time selecting mutated queue item data to obtain a new state of the application under test, even if the coverage does not increase for a long time. Therefore, it is advisable to consider an alternative queue element with better indicators, or which has not yet been investigated, by redirecting the program execution vector.

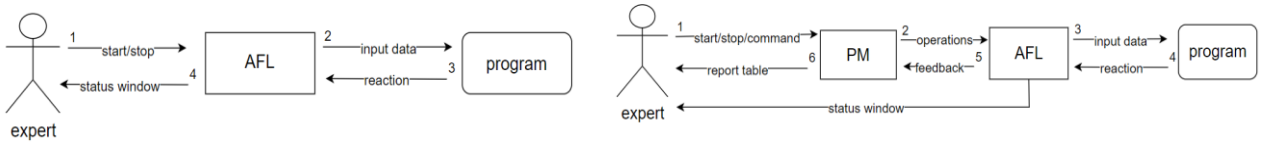
3.1. FUNCTIONAL REQUIREMENTS

In this paper program module integration scheme with AFL is considered, see Figure 2.

The module receives statistics from AFL, analyzes it and shows it to an expert who decides whether to continue working with the current element or with another element of the queue. The module receives this command and writes the corresponding changes to the AFL control files and variables. Then the fuzzing process continues and after a certain period the cycle repeats.

PM gets access to the file traces.txt shared with AFL, in which AFL records traces in the form of tuple sequences and the multiplicity of passing through the tuple within a single execution. After a given period T, the module stops the afl-fuzz process and counts duplicate traces, forming groups. A queue element is defined for each group. The group is associated with information about the coverage share and the number of runs. A report is generated for each element of the queue, which is provided to the expert. The expert analyzes the PM report and the AFL status window and sends a command to the module. To continue the fuzzing process, the module sends a SIGCONT signal to the process and after a period T the cycle repeats. Otherwise, the module changes the queue order. The expert-selected element replaces the current element under study, and the previous set of elements occupies a position in front of it, as already tested.

Table 2 shows a set of basic functions that must be implemented in PM to meet the requirement.



a) *b)*

Figure 2 AFL interaction scheme: a) without program module (PM) and b) with it

Table 2. The proposed tool: the main module functions and their descriptions.

Function Name	Description
getAflPid()	Returns *afl-fuzz* process id
runAflFuzzing()	Runs *afl-fuzz* process
createStateTable()	Create report table for the expert
updateStateTable()	Reports table update
doNextCom()	Performs the next user command
getTotalList()	Analyzes traces set from fuzzer. Returns data structure - list
startNewProcess()	Changes queue and run a new process
printMenu()	Displays command interface menu for the expert

3.2. CONFIGURATIONS AND DEPENDENCIES

Since AFL requires a Linux-based platform, development and configuration is done in the *Ubuntu Desktop 20.04 LTS* distribution. To interact with PM, it is necessary to instrument the source code in the *afl-fuzz.c* file so that the fuzzer records all traces and the multiplicity of tuple execution in the file. Therefore, the *has_new_bits()* function was chosen, which intercepts the new state after the last run. This function is called at each iteration of the loop after the execution of the program. The function updates the *trace_bits[]* array, in which the value of the element equal to 1 corresponds to the label of the tuple included in the current trace, see Figure 3. After instrumentation and code compilation *afl-fuzz* utility is ready to use.

3.3. PROGRAM DEVELOPED STAGE

The programming language **Python 3 version 3.9** was chosen to implement the module. The module code is written in the form of a python script **fuzz.py**. The run command is similar to AFL. After AFL process start, the user receives feedback in the form of a status window and a report table *fuzz.py*, see Figure 4.

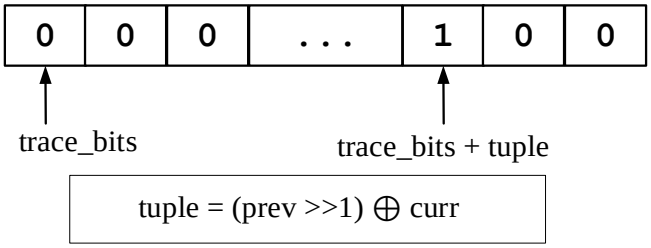


Figure 3 Trace_bits[] array structure

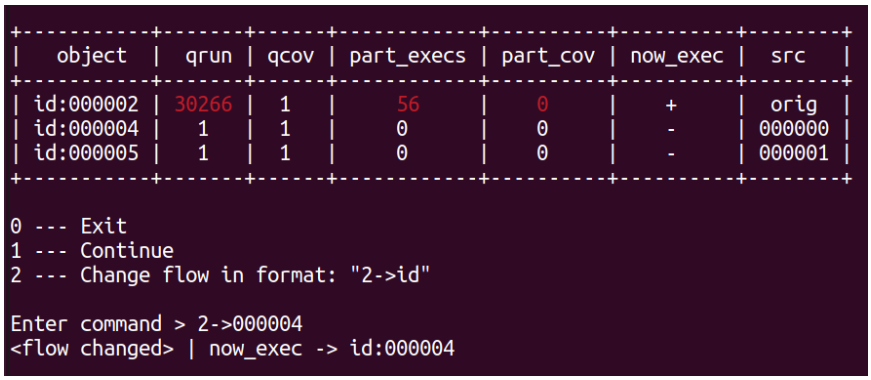


Figure 4 Report table from fuzz.py

The user interacts with the module through the command interface. The main field of the table is **part_cov**. The value of this variable is calculated as the ratio of the number of new unique tuples found **qcov** to the number of runs on the current **qrun** element. This is called *coverage productivity*. **Part_execs** is defined as the program runs percentage on the current element of the total number of runs of the program. The Figure 5 shows a detailed fuzzing-system components interaction.

Receiving the run command from the user, *fuzz.py* sends control signals to the fuzzer, starting the *afl_fuzz* process with certain parameters and report table update time $\mathbf{T_{RT}}$. During this time, the fuzzer performs a standard cycle: retrieves the next element from the queue, mutates it, tests the program response, updates the queue and displays a status window for the expert. At the same time, the specially instrumented *has_new_bits()* function writes traces. When time $\mathbf{T_{RT}}$ is over, *fuzz.py* stops the process, reads the traces, and determines such characteristics as coverage productivity **part_cov**, **part_exec, qcov, qrun**. The results are represented in the form of the report table. The operator analyzes the data in the report table and sends a command to resume the process or redirect the flow if, for example, the *coverage productivity* is small, but at the same time the part_exec value is high. You can redirect the execution flow by selecting an element from the candidate list in the report table. The candidate list was defined by the module at the previous stage. These are queue elements that have not been tested yet or have comparatively better characteristics. At the same time, unexplored elements have a higher priority since they can potentially increase coverage. Having selected an element from the list, the operator sends a command to the module, which modifies the queue order (new element should be in the first place) and makes appropriate changes for the fuzzer service variables to avoid conflict. Finally, file with traces is cleared, the afl_fuzz process resumes, then the cycle repeats again.

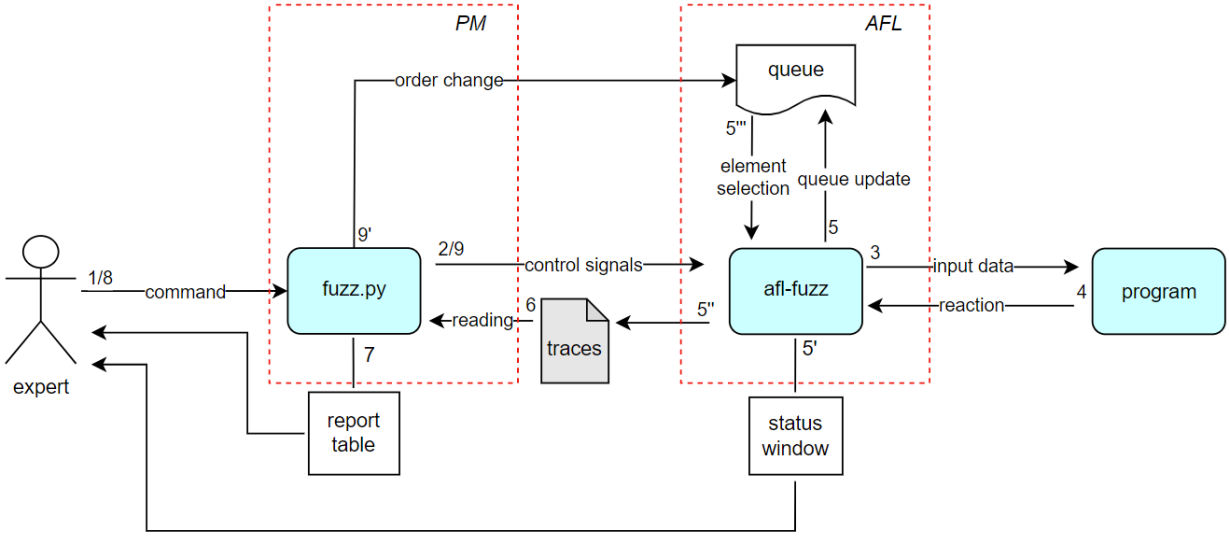


Figure 5 Detailed fuzzing-system components interaction scheme

4. RESULTS

The developed module was checked using the tiff-4.0.3 library utilities, which are designed to work with .tiff files. It is known that many utilities of this library contain a large number of vulnerabilities (Begaev, 2020). Therefore, this library is quite suitable for testing the module and comparing it with other AFL-like fuzzers. AFL and aflFast were chosen as such phasers.

The fuzzing time is T_f = 12 h. This is not enough to have a quality program fuzzing, but it is quite enough to test the developed module, since the utilities have already been tested by the developers. The report table update time T_{RT} = 30 minutes. The fuzzing results are shown in Table 3 and Figure 6. For each utility and each fuzzer, the total number of **unique crashes** and **total coverage** were determined. Having analyzed the results of the table, we can conclude that the proposed fuzzing system in most cases finds crashes faster than other fuzzers.

Comparing with AFLplusplus (AFL++). AFL++ implements a similar mechanism that uses AFLFast module to analyze and process similar code sections during fuzzing. The developed module has been compared with AFLFast project, see Table 3 and Figure 6. The designed module (AFL+ fuzz.py) is able to find more crashes and bugs in less time as well as increasing code coverage.

5. CONCLUSION

In this paper, a new human-controlled fuzzing based on AFL is proposed. This allows to make the fuzzing process more manageable and flexible. The results of the fuzzing system tests and their comparison with other AFL-like fuzzers allow us to conclude that the speed of searching for unique emergency traces has been increased. Also, the total code coverage is greater than the popular AFL implementation shows.

It is important to note that separate research can be carried out by determining the dependence of the fuzzing results on such parameters as the report table update time T_{RT} and the relative number of iterations without changing the fuzzer state Q. The second parameter defines the condition for redirecting the execution flow by the expert. It correlates with coverage productivity. In this work this parameter is defined as $Q \geq 1/3$.

In the future, it is planned to expand the functionality of the fuzzing system by visualizing the control flow graph (CFG) and displaying statistics on the graph edges to the expert.

It is also planned to use this fuzzing system together with static code analyzers to specify and reduce the attack surface.

Analysis of applying machine learning technique for improving fuzzing capabilities will be the subject new research.

Table 3. Fuzzing results.

Library Name	AFL		AFLFast		AFL+ fuzz.py	
Giff2tiff	98	32%	135	68%	178	57%
Ppm2tiff	31	44%	54	52%	73	39%
Tiff2pdf	175	53%	193	71%	214	64%
Jpegopt	27	20%	40	29%	57	31%

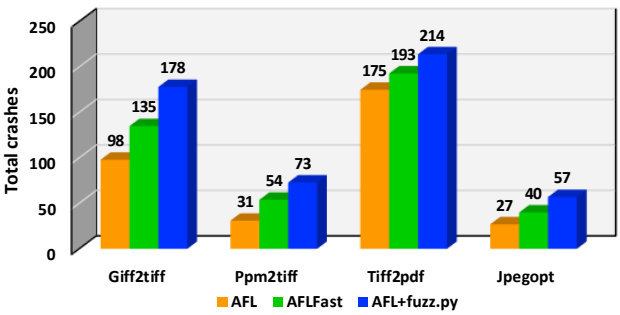


Figure 6 Fuzzing results

6. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback on this work.

7. REFERENCES

[1] Zalewski, M (2020). American Fuzzy Lop. Retrieved from https://lcamtuf.coredump.cx/afl/

[2] AFLfast. (2020). AFLFast (extends AFL with Power Schedules). Retrieved from https://github.com/mboehme/aflfast

[3] libTiff-4.0.3. (2016). Chapter 10. Graphics and Font Libraries. Retrieved from https://linuxfromscratch.org/blfs/view/7.10/general/libtiff.html

[4] Pramanik, A., Tayade, A. (2017). Study and Comparison of General Purpose Fuzzers. University of Wisconsin-Madison, 2017, 19 p.

[5] Mishechkin, M. (2017). Overview of various fuzzing tools as dynamic software analysis tools. Retrieved from https://moluch.ru/archive/186/47575/

[6] Haller, I., Slowinska, A., Neugschwandtner, M.. (2013). A guided fuzzer to find buffer overflow vulnerabilities , , pp 49-64.

[7] Begaev, A., Kashin, S. (2020). Identification of vulnerabilities and undeclared features in software. ITMO University, 2020, 38 p.

[8] Repkin, A. (2020). Condition branch prioritization module for the AFL Automated Vulnerability search system. TUSUR.