

Final Report
AFLRL
American Fuzzy Lop for Reinforcement Learning

Introduction

Fuzz testing, which we even call fuzzing, is one of the automated software testing techniques that have become a cornerstone in the realm of program analysis techniques. It works quite well in injecting a big pile of random, semi-random, or specific input for particular test cases into a program to trigger abnormal behavior, crashes, or memory leaks to identify flaws and vulnerabilities in that software. It was initially developed to uncover security bugs and improve software reliability. Nowadays, fuzzing has evolved to show a wide range of methodologies fitted to tackle the increasing complexity and scalability of modern software systems.

We called such fuzz testing a simple black-box testing approach since at the beginning, the tester pre assumed nothing about the system's internal structure. In the Classical Model, fuzz testing tools generate inputs to feed without knowing the underlying code. They try to test these and check failures through brute-force exploration by Miller et al., 2022. This simplicity allows for fast testing with minimal setup; hence, black-box fuzzing is an easy point at which to start security testing. On the other hand, Zhu et al. (2019) believe that the limitation of black-box fuzzing is such that it cannot prioritize critical paths in the code; it usually can't find deeply embedded bugs because of a lack of context in the input. Fuzzing has evolved from the original simple form of black-box testing to more mature ones like symbolic execution-based fuzzing, vulnerability-oriented fuzzing, and machine learning-based fuzzing. Each of them has pros and cons in coverage and efficiency.

The evolution of symbolic execution techniques with fuzz testing frameworks indeed brings about a sea change as we get better along the way in how we approach the generation of seeds and optimizing inputs. Symbolic execution constructs the test case based on logical expressions to cover certain paths of the program to maximize code outset. This works especially well when comprehensive coverage is required, say in complex multicomponent systems or critical infrastructures. This is where Xie et al. (2020) show how symbolic execution, as applied by their framework these authors called CSEFuzz, improves the efficiency of fuzzing by facilitating simpler test cases and reducing the burden arising from pure randomness in input generation. Symbolic execution has provided one of the most important developments in enabling fuzzers to precisely handle route exploration and identification of vulnerabilities that were not previously possible within big software systems due to computational impracticality for comprehensive path coverage.

Another specialized approach is the one directed to account for vulnerability-oriented fuzzing, giving priority to high-risk system components. In the case of Moukahal et al. (2021), the specific methodology in question focuses on an investigation through their VulFuzz frame copy highlighting security-critical components in CAV systems. VulFuzz focuses the fuzzing process on parts of code that are more likely to hold security vulnerabilities, by assigning higher weights to the more vulnerable sections of code. This approach addresses specific security needs of the

automotive industry while at the same time enhancing the efficiency of fuzz testing and ensuring the most critical parts of the system will be under greater scrutiny. Such domain-specific adjustments of the methodology to certain operational contexts and security requirements can substantially enhance fuzz testing test procedures, such as in the case of vulnerability-oriented fuzzing.

Machine Learning and Artificial Intelligence have been tuned as different models for various techniques and use cases. But the contribution to fuzz testing is to increase the fuzz testing smarter like it plays along with the inputs with different test cases enabling to find more bugs on the software for refinement. Reinforcement learning is a brain of artificial intelligence, said to a lot of research and it comes in handy in fuzzing as a Learning-based fuzzers which can help the higher processing and larger demand for deeper code analysis. It will help reduce the randomness and creation of input flow in different code scenarios for proper testing. This will help to tackle today's complex software code which is very difficult to handle with simple random inputs. Reducing the scope while testing will allow the methodology to work more effectively and efficiently, so Cheng et al. (2024) explain the results from learning-based fuzzers and their betterment over time.

On the other side, Miller et al. (2022) argue basics are important, especially in fuzz testing. Black Box testing is an old method yet it is effective in finding a lot of bugs and serious security flaws on the applications that involve C and C++. Many common issues like memory errors and buffer overflows appear across different programming environments, making these basic fuzzing techniques surprisingly effective. Miller et al. (2022) demonstrated that by using standard fuzzing techniques on Unix utilities, it's possible to reveal vulnerabilities without needing more advanced or specialized tools. Even though a lot of advancement comes in fuzzing or any new methodology, older testing methods still prove their part of finding the vulnerabilities concluding the importance of older frameworks.

Now fuzz testing is evolving based on the arrival of new methods and technologies allowing it to expand its scope for improvement. Each approach brings something valuable, whether it's targeting specific software features, improving performance, or expanding code coverage. These methods come together as a powerful toolset, helping fuzz testing adapt to various software needs—from simple utilities to highly complex autonomous systems. Cheng et al. (2024) and Zhu et al. (2019) both suggest that future research should aim to refine and combine these methods, creating hybrid fuzzing frameworks that make software testing more thorough and efficient.



Fig. 1. Timeline for different fuzzing improvements.

Background and Scope

Fuzzing" or "fuzz testing" is an approach of dynamic program analysis mainly applied to security and robustness testing that feeds unexpected or noise-injected inputs in search of vulnerabilities. Fuzz testing was first conceived more than thirty years ago and, at first took a very straightforward random technique in generation. Fuzzing has evolved, using methods that have grown in sophistication to augment its effectiveness and efficiency. Examples are machine learning, prioritization of vulnerabilities, and symbolic execution, among others. Fuzz testing has evolved from an integral feature of software security to a flexible technique for everyday usage up to high-stakes fields like industrial control software and autonomous systems.

In general, fuzz testing is a black-box software testing technique that provides pseudo-random inputs to a system under test, simulating unexpected user or environmental interactions. This would allow for wide coverage of a variety of areas with the fuzz test, having very little need to know the internal operations of the target program. Miller et al. (2022) revisit this convention in their study of Unix utilities by pointing out that, despite more advanced and modern methods of fuzzing becoming available, old black-box fuzzing remains an effective, approachable choice for finding fundamental vulnerabilities. Black-box fuzzing remains practical, despite the limitations in depth and coverage, for applications where there is a need for fast and inexpensive ways to perform vulnerability testing.

On the other hand, because of the increasing complexity of today's software systems, more sophisticated fuzzing techniques have been developed. One of the most important enhancements in the field of fuzzing is the usage of symbolic execution, which permits the fuzzer to feed the program with tailored test cases, driven by logical expressions from the internal routes of the program. Xie et al. propose CSEFuzz, a fuzzing framework that uses symbolic execution to maximize path coverage of generated test cases. Generally speaking, symbolic execution-based fuzzing is considered a gray-box technique that leverages incomplete program structure knowledge to allow for a more precise and efficient path exploration concerning traditional black-box techniques. In complex security-critical systems where high coverage is needed, symbolic execution plays an extremely vital role, as it raises the possibility of finding hard-to-reach vulnerabilities and centers on a certain part of the branches.

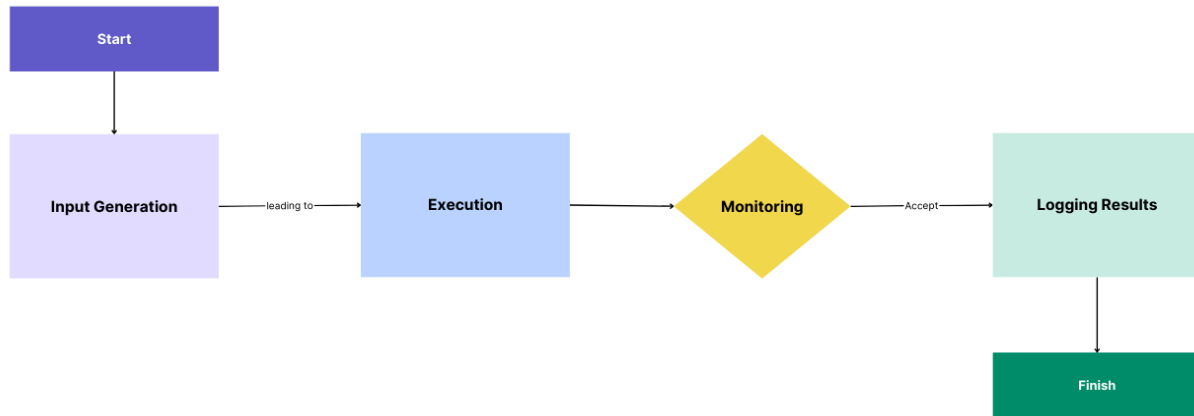


Fig. 2. Fuzzing Process.

Further, domain-specific domains have also adopted fuzz testing where the traditional methods may fail to handle specific operational issues. For example, the participation of a large number of subsystems and protocols for communication creates security challenges for the automotive industry, which is highly dependent on software-based autonomous and connected car systems. As indicated by Moukahal et al. (2021), VulFuzz is a vulnerability-oriented fuzz testing framework targeting CAV systems. In this regard, through appropriate weight allocation concerning the calculated vulnerability metrics on the various software components comprising the system, the discovery of crucial security problems can be maximized while focusing the fuzzing resources on the most vulnerable parts. This domain-specific adaptation of fuzz testing reflects a growing trend towards specialized fuzzing frameworks that are tailored to the security needs of diversified businesses. VulFuzz represents how fuzzing could become one of the specialized methods that can meet unique needs in high-risk sectors like the autonomous vehicle industry, further expanding from its initial usage as a general-purpose testing tool.

Other recent advances in fuzz testing include combinations of machine learning and deep learning-based approaches to further improve test efficiency and adaptability. In presenting the advances in the field, Cheng et al. discuss how machine learning can be used to optimize a range of different aspects of the fuzzing process, including but not limited to scheduling, mutation tactics, and generating seeds. LBA leverages machine learning algorithms' powers of self-improvement in the generation of better inputs and ranking of test cases according to real-world feedback from the tested software. This becomes particularly helpful in scenarios where traditional fuzzing would either be ineffective or expensive because learning-based fuzzing might dynamically adapt to the structure and behavior of the particular program at runtime. According to Cheng et al. (2024), fuzzing of huge and complex systems beyond full coverage by random and symbolic methods may greatly benefit from machine learning-enhanced fuzz testing.

There is always a trade-off between completeness and efficiency, especially with fuzz testing research. Traditional black-box fuzzing has been a fast, light way to identify vulnerabilities but sometimes may lack the depth required for complicated systems. On the other hand, machine learning-enhanced fuzzing and symbolic execution produce even better depth and accuracy,

usually at the expense of more sophisticated setups with manifold processing power. Miller et al. (2022); Xie et al. (2020) In alleviating this challenge, Zhu et al. (2019) developed FEData, a feature-oriented corpus, which enabled the evaluation of a fuzzer by introducing search-impeding features inhibiting state-of-the-art fuzzers from boasting high bug discovery rates. FEData fills the gap between traditional fuzzing and resource-intensive techniques by improving the context awareness of the fuzzer to provide a realistic setting for the evaluation. Zhu et al. say, "The approach provides a way forward for hybrid fuzzing tactics, where contextual improvement can enhance fuzzing without paying the hefty price of advanced techniques."

This review focuses on recent developments in fuzz testing within the last five years by considering how symbolic execution, domain-specific adaptations, and machine learning have been integrated into context-aware corpora. In each of these methods, a set of deficiencies from traditional black-box fuzzing is to be made up for, and the increasing complexity and diversity within the software systems in use today are to be accommodated. The following suites of experiments epitomize how general-purpose and focused the security tool of fuzz testing is, ranging from a variety of fuzzing applications in both specialized and security-critical settings. This review provides a basic understanding of the present state of fuzz testing and its applications by discussing the efficiency, scope, and limitations of these approaches. It sets the foundation for future research in making more intelligent, scalable, and hybrid fuzzing solutions across a wide range of changing software landscapes.

Methodology

A systematic search and selection methodology has been carried out regarding the literature analysis, focusing on the identification of important developments around fuzz testing in the last five years. The search for relevant studies was done through various academic resources: IEEE Xplore, ACM Digital Library, and Google Scholar. Key phrases included "fuzz testing," "program analysis," "vulnerability detection," "symbolic execution," and "learning-based fuzzing." Studies included those basic and influential to the area, recent representative papers reflecting the most recent advances in all fuzz-testing approaches.

Selection criteria were targeted at picking articles that introduced or reviewed fuzz testing approaches in various domains and applications, hence a significant contribution to the discipline. Five key works are selected to demonstrate the variety of recent progress: Xie et al. 2020 for symbolic execution integration, Moukahal et al. 2021 for vulnerability-oriented fuzz testing in autonomous systems, Miller et al. 2022 for insight into classic fuzz testing, Cheng et al. 2024 for machine learning-enhanced fuzzing, and Zhu et al. 2019 for feature-oriented fuzzing corpora.

For each study, the methodology adopted, application domain, and contribution to the effectiveness and coverage of fuzz testing were reviewed. Some key issues have been identified from these selected studies, such as the trade-offs of different fuzzing techniques, their computational cost, and their effectiveness in finding bugs in different software settings.

This literature review is therefore performed in the context of this study and gives an extensive view of the existing processes and future likely paths for research on fuzz testing.

Fuzzing Technique	Key Characteristics	Advantages	Limitations
Black-Box Testing	Simple, minimal setup; relies on random input generation; effective for common vulnerabilities	Quick, accessible, cost-effective; and useful for general vulnerability detection	Limited in-depth; misses deeply embedded vulnerabilities
Feature-Oriented Fuzzing	Uses targeted corpora to focus on feature-specific vulnerabilities	Improves fuzzing efficiency and bug detection accuracy in targeted areas	Requires manually curated corpora; additional overhead
Symbolic Execution-based Fuzzing	Generates test cases based on logical expressions; maximizes code coverage	Enhances path coverage and defect detection; suitable for complex software	Computationally expensive; requires extensive setup
Vulnerability-Oriented Fuzzing	Prioritizes high-risk components; tailored for specific domains (e.g., automotive security)	Focuses on critical vulnerabilities; optimizes testing in specialized domains	Less adaptable to other fields; customized for specific domains
Learning-Based Fuzzing	Applies AI for adaptive input generation; improves efficiency and coverage	Adaptive and intelligent; covers broader code areas; reduces redundancy	Dependent on high-quality datasets; may have high computational costs

Table 1. Comparative Table of Fuzz Testing Techniques.

Challenges and Gaps

While fuzz testing changed a lot, there are still several issues and holes regarding various strategies and techniques. The biggest challenge is how to find an effective balance between accuracy and efficiency. While advanced methods such as learning-based fuzzing and symbolic execution do improve path coverage and flaw identification, they are usually impractical for deployment on either large-scale or time-sensitive applications due to their intensive computing requirements. This higher cost of computation makes these approaches difficult to deploy, especially when the situations are resource-constrained—for example, embedded or real-time systems.

The other significant gap is the absence of domain-specific fuzzing frameworks that can be easily adapted for a wide variety of application areas. Even though VulFuzz was designed for linked autonomous vehicles, it cannot easily be applied in other application domains with different security requirements, for example, IoT devices or industrial control systems. It does, however, point out a larger need for fuzzing tools to be more adaptable and tunable—without large redesigns—to cope with certain security issues.

Traditional black-box fuzzing approaches often provide worse bug-finding results because they are not sufficiently depth- and context-aware when working with complex software structures. An approach to dealing with such a problem is taken by Zhu et al. (2019), who designed FEData, a feature-oriented corpus aimed at enhancing the contextual understanding of fuzzers. Nevertheless, the need for humanly preselected corpora introduces additional labor and is not always feasible to perform under every test scenario. The disparity serves to indicate that more automated, sophisticated means of generating test corpora are needed, which are capable of revealing deeply ingrained weaknesses.

Good quality and the availability of data are yet another concern for learning-based fuzz testing. The machine-learning-enhanced fuzzers need good-quality, expensive datasets to train models to adapt to program behaviors. It would imply that, in the absence of strong training data, learning-based fuzzers would perform poorly, and this may result in lesser coverage, while some important vulnerabilities are not found. Since it is based on datasets, scaling learning-based fuzzing techniques would be tricky as many software types exist.

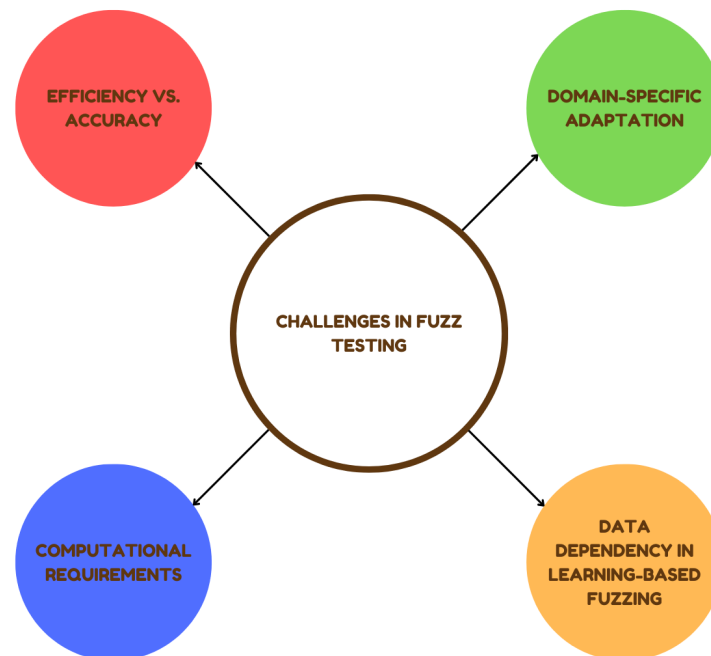


Fig. 3. Challenges in Fuzzing.

Last but not least, despite fixes in fuzzing techniques, the presence of classic software vulnerabilities such as buffer overflows and memory errors indicates that even contemporary software has basic security problems. Traditional fuzzing strategies targeting common software flaws should be prioritized, even while more advanced fuzzing techniques are targeting complicated problems. It, therefore, underlines how much hybrid strategies integrating both traditional and state-of-the-art fuzzing techniques are worth considering to comprehensively address a range of security issues.

These challenges exemplify the remaining significant research and development for fuzzing testing regarding how to develop flexible, resource-efficient methods that can reveal both simple and complex vulnerabilities in a wide range of application areas.

Project Direction

Initially I thought that using the models and learning based fuzzing would improve the accuracy and efficiency of the fuzz testing in general. Because for dynamic analysis like using the ZAP tool for generating reports it will take a lot of time, even days for completion of a complete project.

Fuzzing is no different but taking dynamic analysis to another level of mutating the inputs and testing with the seed, which is the same as slow as dynamic analysis. So I developed the thought of using Reinforcement learning for finding the root cause of the crash that can happen and mutating the input based on that output which is crash here.

This led me to read this paper Xu, D., et al. (2024), which I felt was interesting using the reinforcement learning that rewards the operations involving counterexamples. By balancing random sampling with the exploitation of the counterexamples. This approach was called Racing in their paper which is interesting and an acronym for (Root cAuse analysis on Counter examples based reinforcement learnING).

At first the concept of the mutation on test cases and seeds that possibly cause crash is something helpful for better path exploration and identifying more crashes but training this pattern into the reinforcement learning is highly challenging task and I tried integrating that with the AFL (American Fuzzy Lop) which took unexpected turn on this project as the code base shared by the authors of this paper is not working and producing lot of errors especially version mismatch and environmental problems. The link to the github repository is available here if you want to peek into that, feel free (<https://github.com/0xdd96/Racing-code>).

So the novelty update of the function that I have done to optimize the Racing algorithm failed, so I came up with my idea of training the reinforcement learning using my own dataset. How am I going to create my own dataset? By using the AFL for producing the output(crash data) using the vulnerable code base. Therefore, I changed my topic from **RACING Optimization to “AFLRL” Training Reinforcement Learning models using AFL.**

Therefore the objective of my project for this semester is focussing on producing valuable datasets for training the model, along with AFL has a lot of features and optimizing the model is important, so adding a lot of vulnerable code paths to AFL for producing proper output for training the reinforcement model is my approach.

Below is the flow chart which follows the flow of this project and further discussion about the AFL and its features and discussion on the code is discussed below.

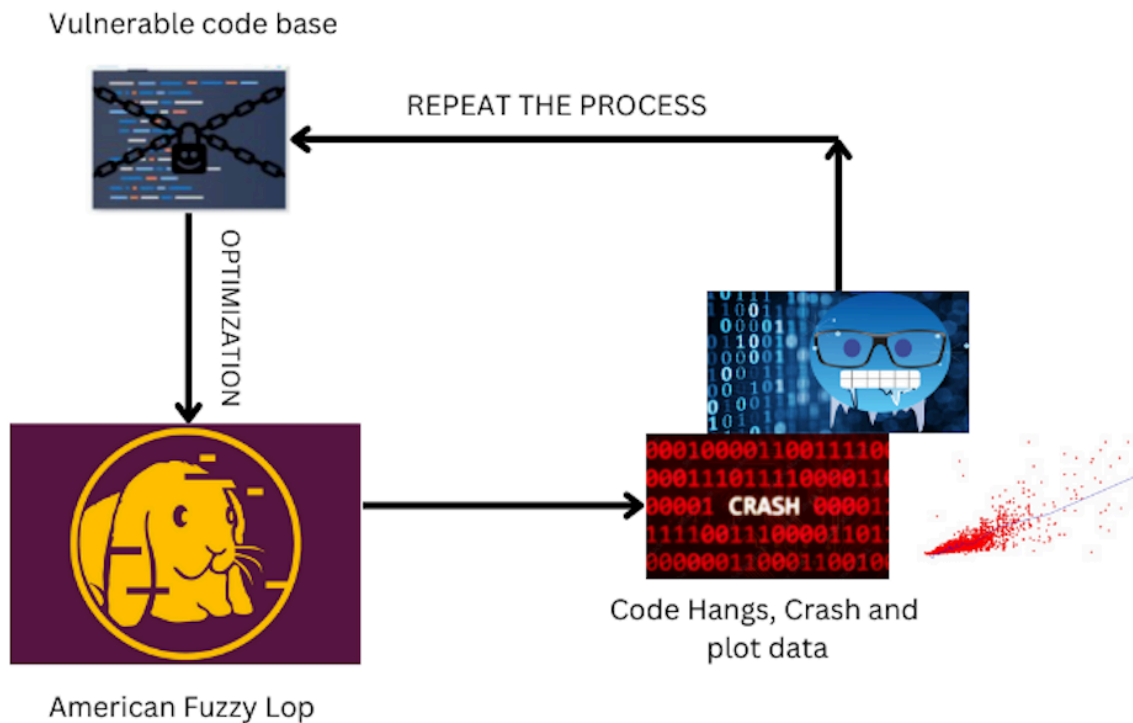


Fig 4. AFLRL Approach

American Fuzzy Lop (AFL)

Introduction

The great majority of remote code execution and privilege escalation defects discovered to date in security-critical software are caused by fuzzing, one of the most effective and tried-and-true methods for spotting security flaws in actual software.

Unfortunately, fuzzing is also rather shallow; some vulnerabilities are firmly outside the scope of this technique since blind, random mutations make it extremely rare to reach specific code pathways in the tested code.

Many attempts have been made to address this issue. Corpus distillation is an early method that was developed by Tavis Ormandy. From a large, high-quality corpus of candidate files, the method uses coverage signals to choose a subset of intriguing seeds, which are then fuzzed using conventional techniques. The method is really effective, but it needs a corpus of this kind to be easily accessible. Furthermore, block coverage metrics are less helpful for long-term fuzzing effort guidance and only offer a very basic insight of program state.

More advanced studies have concentrated on methods like static analysis, symbolic execution, and program flow analysis (also known as "concolic execution"). All these methods are extremely promising in experimental settings, but tend to suffer from reliability and performance problems in practical uses - and currently do not offer a viable alternative to "dumb" fuzzing techniques.

AFL Approach

A brute-force fuzzer with a very straightforward yet incredibly reliable instrumentation-guided genetic algorithm make up American Fuzzy Lop. It effortlessly detects tiny, local-scale changes to program control flow by utilizing a modified version of edge coverage.

To put it simply, the entire algorithm can be summed up as follows:

- Add initial test cases that the user has submitted to the queue.
- Select the subsequent input file from the queue.
- Make an effort to reduce the test case's size such that it doesn't affect the program's measured behavior.
- Alter the file repeatedly using a well-rounded and thoroughly investigated range of conventional fuzzing techniques,
- Add the mutated output as a new entry in the queue if any of the created mutations caused a new state transition that was captured by the instrumentation.
- Proceed to 2.

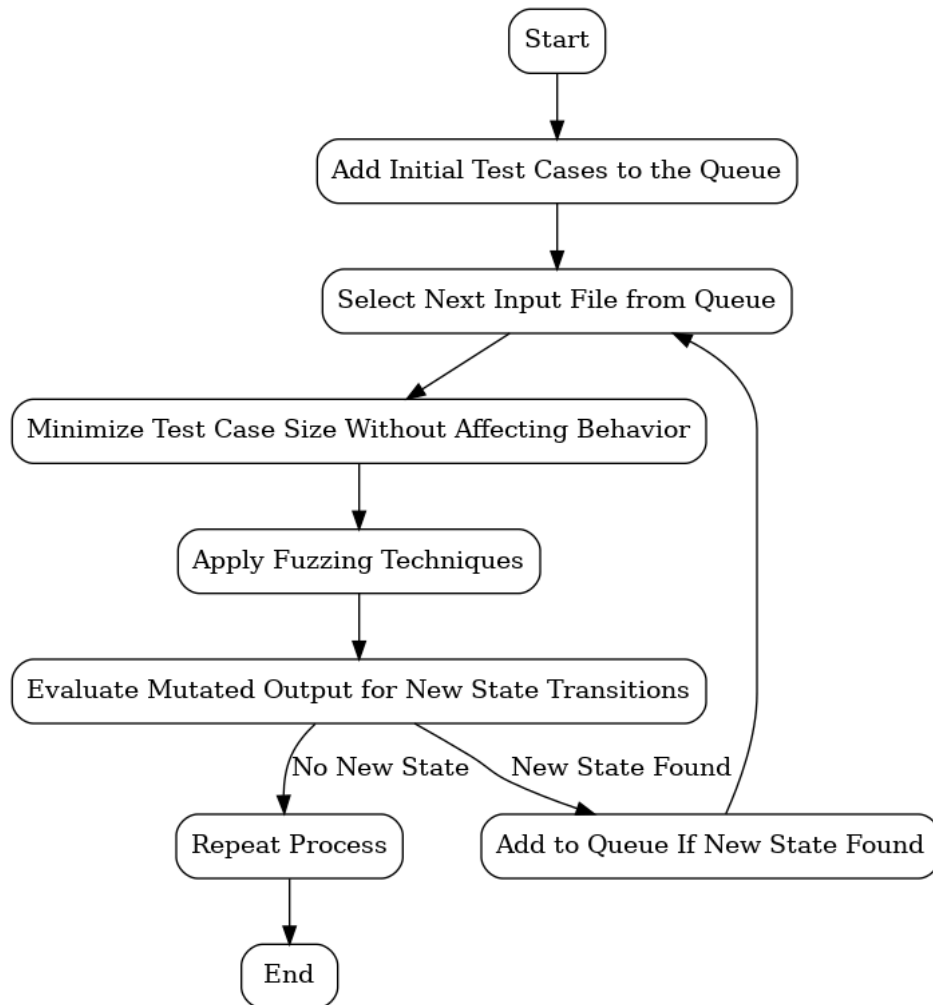


Fig 5. AFL process and it's flow

In addition to going through a number of other instrumentation-driven effort minimization procedures, the found test cases are periodically selected to remove those that have been rendered outdated by more recent, higher-coverage discoveries.

The program produces a small, self-contained corpus of intriguing test cases as a byproduct of the fuzzing process. These are very helpful for seeding other testing regimens that require a lot of work or resources, such stress-testing office programs, graphics suites, browsers, or closed-source products.

According to extensive testing, the fuzzer performs far better out of the box than blind fuzzing or coverage-only tools.

AFL Features

In addition to going through a number of other instrumentation-driven effort minimization procedures, the found test cases are periodically selected to remove those that have been rendered outdated by more recent, higher-coverage discoveries.

The program produces a small, self-contained corpus of intriguing test cases as a byproduct of the fuzzing process. These are very helpful for seeding other testing regimens that require a lot of work or resources, such stress-testing office programs, graphics suites, browsers, or closed-source products.

According to extensive testing, the fuzzer performs far better out of the box than blind fuzzing or coverage-only tools.

```
$ CC=/path/to/afl/afl-gcc ./configure  
$ make clean all
```

For C++ programs, you'd also want to set CXX=/path/to/afl/afl-g++.

The same applies to the clang wrappers (afl-clang and afl-clang++); clang users can additionally choose to utilize a higher-performance instrumentation mode, as explained in [llvm_mode/README.llvm](#).

Finding or creating a straightforward program that receives data from a file or from stdin and sends it to the tested library is necessary for testing libraries. Linking this executable to a static version of the instrumented library or ensuring that the correct.so file is loaded at runtime (often by changing LD_LIBRARY_PATH) are crucial in such a situation. The simplest choice is a static build, which may typically be accomplished by:

```
$ CC=/path/to/afl/afl-gcc ./configure --disable-shared
```

Setting AFL_HARDEN=1 when calling 'make' will cause the CC wrapper to automatically enable code hardening options that make it easier to detect simple memory bugs. Libdislocator, a helper library included with AFL (see libdislocator/README.dislocator) can help uncover heap corruption issues, too.

PS. ASAN users are advised to review [notes_for_asan.txt](#) file for important caveats.

ASAN which stands for Address Sanitizer helps to find the memory error and bugs in C/C++ programs. It works parallel with AFL for finding memory issues and other leaks which AFL can miss sometimes. We have implemented this in our code base.

What is ASAN?

AddressSanitizer is a **runtime memory debugging tool** that instruments the binary to detect various types of memory-related bugs, such as:

- Buffer overflows (heap and stack)
- Use-after-free vulnerabilities
- Memory leaks
- Invalid memory access

It provides detailed information about memory corruption, including stack traces and the exact location of the issue in the source code.

Why Use ASAN with AFL?

1. **Enhanced Bug Detection:** AFL finds out the bugs but ASAN can find out the exact error related to memory.
2. **Detailed Diagnostics:** Stack stases and memory dumps which is generated by ASAN is helpful to developers to make note of the important bugs.
3. **Uncover Hidden Bugs:** Some odd bugs will not produces the crash but can be affect later on the program which can be revealed by ASAN.

Fuzzing Binaries

The afl-fuzz utility does the actual fuzzing process. This program needs a path to the binary to text, a read-only directory containing the initial test cases, and a different location to keep its results.

The standard syntax for target binaries that take input straight from stdin is:

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program [...params...]
```

Use '@@' to indicate in the target's command line where the input file name should be entered for programs that accept input from a file. This is what the fuzzer will replace you with:

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

To have the altered data written to a particular file, you can also use the -f option. This is helpful if the application anticipates a specific file extension.

Non-instrumented binaries can be fuzzed using either the conventional blind-fuzzer mode (specify -n) or the QEMU mode (add -Q in the command line).

Video decoders and compilers are two uncommon examples of targets that can require these settings to be changed. You can use -t and -m to override the executed process's default timeout and memory limit.

It should be noted that afl-fuzz begins by carrying out a series of deterministic fuzzing stages, which can take days but typically result in clean test cases. In the command line, add the -d

option if you want immediate, dirty results similar to those of zzuf and other conventional fuzzers.

Vulnerability Details

1. Use-After-Free in `json_value_free_ex`

Code Block:

```
if (value->u.array.length == 0) {  
    free(*top);  
}
```

- **Issue:**

The code frees memory for a **json_value** object when the array length is zero. However, the freed memory block might still be accessed later in the program. This is a classic **use-after-free** vulnerability.

- **Impact:** This can lead to undefined behavior, including crashes or memory corruption, especially when the freed memory is overwritten by new allocations.
- **Fix:** After freeing, set the pointer to **NULL** to avoid further dereferencing. Alternatively, rework the logic to prevent accessing freed memory.
- **Why It's Useful:** AFL will generate inputs like `[]` to trigger this case, revealing use-after-free issues. These inputs and their effects can be used as training data for models to predict such issues.

2. Invalid Free in JSON Object

Code Block:

```
value = value->u.object.values[value->u.object.length--].value;
```

- **Issue:** The decrement operator `--` is applied **after** the index is used, causing the object length to be incorrectly decremented. As a result, the memory block that is freed is not the expected one.
- **Impact:** Leads to invalid free operations, causing crashes or heap corruption.
- **Fix:** Correct the decrement order to `--value->u.object.length` so that the index decrements before use.
- **Why It's Useful:** Inputs containing objects (e.g., `{"key": 0}`) trigger this bug. AFL can uncover edge cases, such as objects with minimal or maximal keys, to cause invalid memory operations.

3. Invalid Free in Empty String Handling

Code Block:

```
if (!value->u.string.length) {  
    value->u.string.ptr--;  
}  
settings->mem_free(value->u.string.ptr, settings->user_data);
```

- **Issue:** Decrementing the pointer (**ptr--**) before freeing results in an invalid memory reference, leading to undefined behavior.
- **Impact:** The program might crash due to attempting to free memory that it does not own.
- **Fix:** Avoid pointer manipulation on strings before freeing memory.
- **Why It's Useful:** Inputs like "" test the program's behavior for empty strings. AFL identifies memory management issues by mutating inputs to explore this path.

NULL Pointer Dereference

Code Block:

```
if (value->u.string.length == 1) {  
    char *null_pointer = NULL;  
    printf("%d", *null_pointer);  
}
```

- **Issue:** If the string length is one, the code creates and dereferences a **NULL** pointer. This is a critical bug.
- **Impact:** Causes a segmentation fault, crashing the program.
- **Fix:** Remove the dereference of **null_pointer** entirely.
- **Why It's Useful:** A single-character string ("A") causes this crash. AFL's input mutations ensure coverage of such boundary conditions, providing insights into how code handles unusual inputs.

Training Reinforcement Learning

We are collecting good data for training the reinforcement learning through the project by testing the AFL dynamic path exploration capabilities for improving its accuracy and implementing it relatively faster than existing speed. Especially for a larger code base.

The Training process will look like the following:

1. Data Collection:

- Use AFL to generate inputs and capture the resulting crashes or unusual behavior.
- Record input features (e.g., JSON structure, sizes) and crash metadata (e.g., error type, stack trace).

2. Feature Engineering:

- Encode JSON inputs as numerical or structural features.
- Annotate code paths explored by AFL as additional input features.

3. Model Training:

- Use supervised learning to train models to predict crash likelihood or detect anomalies based on input features and execution paths.

4. Testing and Deployment:

- Test the trained model on new JSON inputs to predict vulnerabilities or flag potentially dangerous code paths.

By combining AFL's fuzzing capabilities with neural network training, you can develop models that not only detect existing bugs but also anticipate vulnerabilities in unseen code.

Implementation

We are using the C programming language for this purpose and the complete tutorial and code base is available in my github repository in the following link.

<https://github.com/kp18-cpu/afirl>

In this section I will explain the step by step process of implementing this process of running the AFL on our vulnerable code base and analyse the results.

Docker Installation

We will install docker for using the Ubuntu 20.04 image specifically for successful installation of this program so that we will avoid the version error that may be caused by some libraries on the AFL.

I am going to create an EC2 instance for this purpose but you can use VMware, Vbox, Parallels for this implementation as well.

Before installing the docker, you encounter errors regarding the linux kernel especially in the future so make sure to run the below command.

```
echo core > /proc/sys/kernel/core_pattern
```

```
sudo apt update
sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker
```


Instance summary for i-005ae1f03da9cc784 [Info](#)

Instance details

Updated less than a minute ago

Instance ID

 i-005ae1f03da9cc784

IPv6 address

—

Hostname type

IP name: ip-172-31-17-103.ec2.internal

Answer private resource DNS name

IPv4 (A)


Public IPv4 address

 52.55.101.64 | [open address](#) 

Instance state

 Running

Private IP DNS name (IPv4 only)

 ip-172-31-17-103.ec2.internal

Instance type

t2.micro

```
ubuntu@ip-172-31-17-103:~$ sudo su
root@ip-172-31-17-103:/home/ubuntu# apt update
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu jammy-updates InRelease [128 kB]
Get:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu jammy-backports InRelease [127 kB]
```

```
root@ip-172-31-17-103:/home/ubuntu# apt install docker.io
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap docker-doc rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base docker.io pigz runc ubuntu-fan
0 upgraded, 8 newly installed, 0 to remove and 30 not upgraded.
Need to get 75.5 MB of archives.
After this operation, 284 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

```
root@ip-172-31-17-103:/home/ubuntu# systemctl start docker
root@ip-172-31-17-103:/home/ubuntu# systemctl enable docker
```

Time to pull the image and run it through the docker.

```
docker pull ubuntu:20.04
docker run -it ubuntu:20.04
```

```
root@ip-172-31-17-103:/home/ubuntu# docker pull ubuntu:20.04
20.04: Pulling from library/ubuntu
d9802f032d67: Pull complete
Digest: sha256:8e5c4f0285ecbb4ead070431d29b576a530d3166df73ec44affc1cd27555141b
Status: Downloaded newer image for ubuntu:20.04
docker.io/library/ubuntu:20.04
root@ip-172-31-17-103:/home/ubuntu# docker run -it ubuntu:20.04
root@7f1234b2539e:/#
```

AFL Installation

Installation of AFL is straightforward and we will include the important libraries before pulling the AFL through the github. Update the system first and install the following packages.

```
sudo apt-get update # Make sure to run this command if you get unable to
locate package error.
sudo apt install -y build-essential clang llvm python3 wget git
```

We will clone the repository using the git and look for the make file and start the installation process of the AFL in our ubuntu image.

```
root@7f1234b2539e:/# git clone https://github.com/google/AFL.git
Cloning into 'AFL'...
remote: Enumerating objects: 531, done.
remote: Total 531 (delta 0), reused 0 (delta 0), pack-reused 531 (from 1)
Receiving objects: 100% (531/531), 942.49 KiB | 15.97 MiB/s, done.
Resolving deltas: 100% (196/196), done.
```

```
git clone https://github.com/google/AFL.git
cd AFL
```

What is makefile in C programming?

The **Makefile** in C is very important because it compiles and automates the process of building the project. Especially linking different files with the appropriate paths is critical. It is useful for managing the code base with multiple files and dependencies.

```
root@7f1234b2539e:/AFL# ls
Android.bp      README.md      afl-cmin      afl-plot      alloc-inl.h    dictionaries  libdislocator  test-instr.c
CONTRIBUTING.md afl-analyze.c  afl-fuzz.c    afl-showmap.c android-ashmem.h docs          libtokenencap  test-libfuzzer-target.c
LICENSE         afl-as.c       afl-gcc.c     afl-tmin.c    config.h        experimental  llvm_mode      testcases
Makefile        afl-as.h       afl-gotcpu.c  afl-whatsapp  debug.h         hash.h        qemu_mode      types.h
root@7f1234b2539e:/AFL#
```

Run make and make install to complete the installation of AFL on the system.

```
make
make install
```

```
root@7f1234b2539e:/AFL# make
[*] Checking for the ability to compile x86 code...
[*] Everything seems to be working, ready to compile.
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-gcc.c -o afl-gcc -ldl
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $i; done
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-fuzz.c -o afl-fuzz -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-showmap.c -o afl-showmap -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-tmin.c -o afl-tmin -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-gotcpu.c -o afl-gotcpu -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-analyze.c -o afl-analyze -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" afl-as.c -o afl-as -ldl
ln -sf afl-as as
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" test-instr.c -o test-instr -ldl
./afl-showmap -m none -q -o .test-instr0 ./test-instr < /dev/null
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[*] All right, the instrumentation seems to be working!
[*] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[*] All done! Be sure to review README - it's pretty short and useful.
NOTE: If you can read this, your terminal probably uses white background.
This will make the UI hard to read. See docs/status_screen.txt for advice.
```

```
root@7f1234b2539e:/AFL# make install
[*] Checking for the ability to compile x86 code...
[*] Everything seems to be working, ready to compile.
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH="/usr/local/lib/afl" -DDOC_PATH="/usr/local/share/doc/afl" -DBIN_PATH="/usr/local/bin" test-instr.c -o test-instr -ldl
./afl-showmap -m none -q -o .test-instr0 ./test-instr < /dev/null
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[*] All right, the instrumentation seems to be working!
[*] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[*] All done! Be sure to review README - it's pretty short and useful.
NOTE: If you can read this, your terminal probably uses white background.
This will make the UI hard to read. See docs/status_screen.txt for advice.
mkdir -p -m 755 ${DESTDIR}/usr/local/bin ${DESTDIR}/usr/local/lib/afl ${DESTDIR}/usr/local/share/doc/afl ${DESTDIR}/usr/local/share/afl
rm -f ${DESTDIR}/usr/local/bin/afl-plot.sh
install -m 755 afl-gcc afl-fuzz afl-showmap afl-tmin afl-gotcpu afl-analyze afl-plot afl-cmin afl-whatsapp ${DESTDIR}/usr/local/bin
rm -f ${DESTDIR}/usr/local/bin/afl-as
if [ -f afl-qemu-trace ]; then install -m 755 afl-qemu-trace ${DESTDIR}/usr/local/bin; fi
if [ -f afl-clang-fast -a -f afl-llvm-pass.so -a -f afl-llvm-rt.o ]; then set -e; install -m 755 afl-clang-fast ${DESTDIR}/usr/local/bin; ln -sf afl-clang-fast ${DESTDIR}/usr/local/bin/afl-clang-fast++; install -m 755 afl-llvm-pass.so afl-llvm-rt.o ${DESTDIR}/usr/local/lib/afl; fi
if [ -f afl-llvm-rt-32.o ]; then set -e; install -m 755 afl-llvm-rt-32.o ${DESTDIR}/usr/local/lib/afl; fi
if [ -f afl-llvm-rt-64.o ]; then set -e; install -m 755 afl-llvm-rt-64.o ${DESTDIR}/usr/local/lib/afl; fi
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc ${DESTDIR}/usr/local/bin/$i; done
install -m 755 afl-as ${DESTDIR}/usr/local/lib/afl/as
ln -sf afl-as ${DESTDIR}/usr/local/lib/afl/as
install -m 644 README.md docs/ChangeLog docs/*.txt ${DESTDIR}/usr/local/share/doc/afl
cp -r testcases/ ${DESTDIR}/usr/local/share/afl
cp -r dictionaries/ ${DESTDIR}/usr/local/share/afl
root@7f1234b2539e:/AFL#
```

Type the following command to check whether the AFL is successfully installed:

```
afl-fuzz -V
```

```
root@7f1234b2539e:/AFL# afl-fuzz -V
afl-fuzz 2.57b by <lcamtuf@google.com>
```

There are a lot of tunings and folders can be edited for the different use case of AFL like Qemu and afl-clang, afl-gcc.

Cloning AFLRL Repository

The C program has been specifically developed and modified to include deliberate backdoors and multiple memory corruption vulnerabilities, aiming to enhance the efficacy of fuzzers and analysis tools. Each vulnerability is carefully documented in aflrl.c, with corresponding input files under input-files/ designed to trigger them. By incorporating these vulnerabilities, the code provides a robust framework for training AFL to adopt reinforcement learning strategies, enabling better mutation techniques and leveraging counterexamples for improved fuzzing in future scenarios.

```
git clone https://github.com/kp18-cpu/aflrl.git
```

```
root@7f1234b2539e:/# git clone https://github.com/kp18-cpu/aflrl.git
Cloning into 'aflrl'...
remote: Enumerating objects: 36, done.
remote: Counting objects: 100% (36/36), done.
remote: Compressing objects: 100% (28/28), done.
remote: Total 36 (delta 12), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (36/36), 20.15 KiB | 793.00 KiB/s, done.
root@7f1234b2539e:/# cd aflrl
root@7f1234b2539e:/aflrl#
```

Install all the dependencies and handle the package using the make command as it is a C code.

```
make
```

```

root@7f1234b2539e:/aflrl# make
afl-gcc -o aflrl -I. main.c aflrl.c -lm
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 75 locations (64-bit, non-hardened mode, ratio 100%).
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 375 locations (64-bit, non-hardened mode, ratio 100%).
afl-gcc -fsanitize=address -o aflrl_ASAN -I. main.c aflrl.c -lm
afl-cc 2.57b by <lcamtuf@google.com>
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 40 locations (64-bit, ASAN/MSAN mode, ratio 33%).
afl-as 2.57b by <lcamtuf@google.com>
[+] Instrumented 354 locations (64-bit, ASAN/MSAN mode, ratio 33%).
root@7f1234b2539e:/aflrl# █

```

Now we are ready to execute the afl on our c code for finding the crashes, exploring the code paths etc, we will analyse the results later. Run either of the command below to invoke the AFL.

```
make afl
```

```
----or----
```

```
afl-fuzz -i in -o out ./aflrl @@
```

Once the AFL starts, wait for 2 to 5 min for getting enough path exploration and crashes.

american fuzzy lop 2.57b (aflrl)			
process timing		overall results	
run time : 0 days, 0 hrs, 9 min, 37 sec		cycles done : 2	
last new path : 0 days, 0 hrs, 1 min, 2 sec		total paths : 316	
last uniq crash : 0 days, 0 hrs, 0 min, 46 sec		uniq crashes : 20	
last uniq hang : none seen yet		uniq hangs : 0	
cycle progress		map coverage	
now processing : 165* (52.22%)		map density : 0.10% / 0.75%	
paths timed out : 0 (0.00%)		count coverage : 2.69 bits/tuple	
stage progress		findings in depth	
now trying : interest 32/8		favored paths : 86 (27.22%)	
stage execs : 1677/1703 (98.47%)		new edges on : 121 (38.29%)	
total execs : 1.24M		total crashes : 1449 (20 unique)	
exec speed : 2147/sec		total tmouts : 0 (0 unique)	
fuzzing strategy yields		path geometry	
bit flips : 24/17.9k, 9/17.7k, 8/17.3k		levels : 15	
byte flips : 0/2232, 0/2058, 1/1729		pending : 143	
arithmetics : 41/124k, 0/19.6k, 0/1453		pend fav : 0	
known ints : 2/11.9k, 2/56.1k, 0/74.0k		own finds : 315	
dictionary : 0/0, 0/0, 0/0		imported : n/a	
havoc : 248/893k, 0/0		stability : 100.00%	
trim : 26.86%/607, 0.00%			
[cpu:711%]			

Stop the fuzzing the process through Ctrl + c or Control + c.

Navigate to the out folder and look for the subfolders which has the results and details of the fuzz testing which we just ran.

```
root@56264d19151b:/aflrl# cd out
root@56264d19151b:/aflrl/out# ls
crashes fuzz_bitmap fuzzer_stats hangs plot_data queue
root@56264d19151b:/aflrl/out#
```

Below we can see the mutated input that caused the crash and the ID 000012 means 12th mutated input caused the crash.

```
root@56264d19151b:/aflrl/out/crashes# cat id:000018,sig:11,src:000303,op:havoc,rep:2
"\c"root@56264d19151b:/aflrl/out/crashes# cat id:000012,sig:11,src:000016,op:arith8,pos:5,val:+5
{"":-,}
root@56264d19151b:/aflrl/out/crashes# id:000008,sig:06,src:000003,op:arith8,pos:3,val:+35
bash: id:000008,sig:06,src:000003,op:arith8,pos:3,val:+35: command not found
root@56264d19151b:/aflrl/out/crashes# cat id:000008,sig:06,src:000003,op:arith8,pos:3,val:+35
[""]root@56264d19151b:/aflrl/out/crashes#
```

The Queue folder has all the input in queue and different types of mutated input for fuzz testing.

```
[~]root@56264d19151b:/aflrl/out/queue# cat id:000217,src:000211,op:havoc,rep:2  
{"oooooooo\uc7cd0@o\uc7cd0}root@56264d19151b:/aflrl/out/queue# cat id:000218,src:000212,op:arith8,pos:8,val:+1,+cov  
"0Zoo\udcc7root@56264d19151b:/aflrl/out/queue# █
```

Finally we will look into the plot values that this AFL is generated on the code we wrote.

```
# unix_time, cycles_done, cur_path, paths_total, pending_total, pending_favs, map_size, unique_crashes, unique_hangs, max_depth, execs_per_sec  
1733178821, 0, 0, 1, 1, 1, 0.12%, 0, 0, 1, 1000.00  
1733178826, 0, 0, 93, 93, 1, 0.46%, 5, 0, 2, 1915.24  
1733178831, 0, 0, 106, 106, 1, 0.48%, 7, 0, 2, 2120.14  
1733178836, 0, 0, 114, 114, 1, 0.48%, 8, 0, 2, 2154.39  
1733178842, 0, 3, 121, 120, 44, 0.49%, 10, 0, 3, 2130.90  
1733178847, 0, 3, 124, 123, 44, 0.50%, 11, 0, 3, 2180.30  
1733178852, 0, 3, 125, 124, 44, 0.50%, 11, 0, 3, 2146.02  
1733178857, 0, 16, 127, 122, 43, 0.51%, 14, 0, 3, 2179.03  
1733178862, 0, 22, 129, 118, 39, 0.52%, 14, 0, 3, 2204.30  
1733178867, 0, 35, 130, 113, 34, 0.52%, 15, 0, 3, 2213.23  
1733178872, 0, 38, 141, 122, 32, 0.53%, 15, 0, 3, 2208.53  
1733178877, 0, 38, 144, 125, 32, 0.53%, 15, 0, 3, 2176.67  
1733178883, 0, 45, 145, 120, 29, 0.53%, 15, 0, 3, 2195.86  
1733178888, 0, 49, 153, 127, 28, 0.53%, 15, 0, 3, 2187.73  
1733178898, 0, 59, 159, 130, 24, 0.54%, 15, 0, 3, 2170.56  
1733178903, 0, 59, 167, 138, 24, 0.54%, 15, 0, 3, 2189.79  
1733178908, 0, 59, 169, 140, 24, 0.54%, 15, 0, 3, 2163.07  
1733178918, 0, 71, 172, 138, 23, 0.55%, 15, 0, 3, 2137.40  
1733178923, 0, 112, 174, 134, 17, 0.55%, 15, 0, 3, 2162.09  
1733178928, 0, 130, 178, 132, 14, 0.56%, 15, 0, 4, 2154.56  
1733178933, 0, 130, 184, 138, 14, 0.56%, 15, 0, 4, 2137.44  
1733178938, 0, 130, 185, 139, 14, 0.57%, 15, 0, 4, 2130.54  
1733178943, 0, 151, 191, 143, 14, 0.57%, 15, 0, 4, 2075.39  
1733178948, 0, 151, 192, 144, 14, 0.57%, 15, 0, 4, 2130.38  
1733178954, 0, 152, 195, 146, 14, 0.57%, 15, 0, 4, 2166.33  
1733178959, 0, 156, 196, 145, 12, 0.57%, 15, 0, 4, 2126.51  
1733178964, 0, 158, 200, 147, 12, 0.58%, 15, 0, 4, 2128.05
```

This is how we can execute the fuzzy lop for finding the code crash and results on mutated inputs.

Challenges

This is a great way of mutating the inputs and triggering the bugs for finding the crashes, but the data we extracted are still very hard to preprocess for training the Reinforcement model. It requires a good pipeline that can be built through the Jenkins for optimization and storing of the data for further analysis.

The future work of this project can be enhanced properly with the CI/CD Devsecops pipelines for knowing the possibilities of crashes rather than collecting it.

This test can be performed for code check and bug finding and enhancing it with a vulnerable code base is a win for training models especially.

Conclusion

Fuzz testing has shown itself to be a robust and agile approach in program analysis in general, especially in vulnerability discovery and software security hardening. Methodologies have evolved from traditional black-box issues toward more sophisticated symbolic execution, large domain-specific frameworks, and learning-based models. Through feature-oriented corpora, symbolic execution-enhanced seed generation, and AI-driven adaptability, fuzz testing has expanded to tackle increasingly complex software environments and security challenges (Zhu et al., 2019; Xie et al., 2020; Moukahal et al., 2021; Cheng et al., 2024; Miller et al., 2022).

However, it also leaves considerable challenges and gaps in the areas such as balancing computational efficiency with comprehensive coverage, developing domain-specific fuzzing frameworks adaptable, and enhancing fuzzing contextual awareness. Moreover, the dependence of learning-based fuzzing on high-quality datasets further expands the complexity of scaling these approaches across diverse software applications. While these improvements are taking place, findings of traditional vulnerabilities still come up, which thereby justifies the hybrid methods in modern times, merging classic and advanced fuzzing techniques that would ensure the checking of shallow and deep vulnerabilities.

In the future, research should focus on developing hybrid approaches, effectively merging traditional, symbolic, and AI-driven fuzzing in order to take the area of fuzz testing one step forward. A very important aspect here is enriching the training in vulnerable code patterns for the RL models. This specific approach will ensure the optimization of fuzzing frameworks, such as AFL and many others, by better gathering, refinement, and quality, ensuring effective efficiency in the generated solutions.

Furthermore, this requires flexible and resource-efficient fuzzing frameworks that can meet different and constantly changing requirements from various industries. Reinforcement learning methods with real-world vulnerable code datasets can help in runtime, complex attack vector identification, and scalability of depth in software vulnerability analysis. Using the optimization heuristics of RL, fuzzers will become intelligent in guiding test case generation, prioritizing on critical paths, and also minimizing redundancies in a testing cycle.

In such a manner, advances will continue to bring about further evolution of the fuzz testing technique—a necessary tool to reach strong security in software. Therefore, future fuzzing techniques may be used for increasingly complex technological landscapes, while being sustained towards the specific needs of the industries involved to assure effectiveness, scalability, and thoroughness. A framework combining these RL-driven methods with more traditional and symbolic techniques constitutes an end-to-end proactive, efficient detection solution of vulnerabilities.

References

1. Zhu, X., Feng, X., & Jiao, T. (2019). A Feature-Oriented Corpus for Understanding, Evaluating and Improving Fuzz Testing. *ACM*.

Cited by 17

2. Xie, Z., Cui, Z., & Zhang, J. (2020). CSEFuzz: Fuzz Testing Based on Symbolic Execution. *IEEE*.

Cited by 6

3. Moukahal, L., Zulkernine, M., & Soukup, M. (2021). Vulnerability-Oriented Fuzz Testing for Connected Autonomous Vehicle Systems. *IEEE Transactions*.

Cited by 35

4. Miller, B., Zhang, M., & Heymann, E. (2022). The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions*.

Cited by 42

5. Cheng, H., Li, D., & Zhao, M. (2024). A Comprehensive Review of Learning-Based Fuzz Testing Techniques. *IEEE Symposium*.

Cited by 2

6. Xu, Dandan, et al. "Racing on the Negative Force: Efficient Vulnerability {Root-Cause} Analysis through Reinforcement Learning on Counterexamples." 33rd USENIX Security Symposium (USENIX Security 24). 2024.

Cited by 1

7. Oliinyk, Yaroslav, et al. "Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing." *arXiv preprint arXiv:2403.03897* (2024).

Cited by 8

Link 1:

https://ieeexplore.ieee.org/abstract/document/10562114/?casa_token=ZhZiJNEF_MAAAAA:T N7idy_Zt0ASC4li90BOZvMSbpcd95gA2Cj49qDrd8BNerH7SiaHXGvbWSHtOp1W6vcTzE7PnA

Link 2:

https://ieeexplore.ieee.org/abstract/document/9309406?casa_token=bB0hcBOP4KIAAAAA:xub VuloXXAZrfJoO-2mv_pwo3KefkAuD7-7qtis-qURtVVcwEBWKzeCnS_8_Asui0otqBjdhHg

Link 3:

https://ieeexplore.ieee.org/abstract/document/9557794?casa_token=uJZDDGnkot4AAAAA:Xo9 ZC5AmFPDbBVAXpOcSTyw4xYbz1h0O-lI1bHqGpHD213ZMVvKMG4vasoalUd6Uf0G8bzkckA

Link 4:

<https://ieeexplore.ieee.org/abstract/document/9222017>

Link 5:

https://dl.acm.org/doi/abs/10.1145/3321705.3329845?casa_token=m8S0TzCTv_EAAAAA:FMvHmkrBLntwdrsT4xePa1maKVStkuuAHAnFWY4bWo7jOo7Jbmd44i6y8Qtyg0NIIAIJZwA2NcDU

Link 6:

<https://www.usenix.org/conference/usenixsecurity24/presentation/xu-dandan>

Link 7:

<https://www.usenix.org/conference/usenixsecurity24/presentation/asmita>

Keyword lists

1. Fuzz Testing
2. Program Analysis
3. Security Vulnerability Detection
4. Black-Box Fuzzing
5. Symbolic Execution
6. Vulnerability-Oriented Fuzzing
7. Learning-Based Fuzzing
8. Automated Software Testing
9. Machine Learning in Fuzzing
10. Path Coverage Optimization
11. Adaptive Testing Frameworks
12. Feature-Oriented Corpus
13. AI-Driven Fuzzing Techniques
14. Input Mutation Strategies
15. Dynamic Program Analysis
16. Computational Efficiency in Testing
17. Code Coverage Maximization
18. Domain-Specific Fuzzing
19. Connected Autonomous Vehicles Security
20. Hybrid Fuzz Testing Techniques