Chapter 2

Background

2.1 Introduction

Fuzzing (short for fuzz testing) is a tool for "finding implementation bugs using malformed/semi-malformed data injection in an automated fashion" . Since the late '80s, fuzz testing has proved to be a powerful tool for finding errors in a program. For instance, American Fuzzy Lopper (AFL) has found more than a total of 330 vulnerabilities from 2013 to 2017 in more than 70 different programs [5]. The research on fuzz testing has found its place in software security testing. Liang et al. [1] illustrates the growth of the primary studies from the following publishers: *ACM digital library*, *Elsevier ScienceDirect*, *IEEEXplore digital library*, *Springer online library*, *Wiley InterScience*, *USENIX*, and *Semantic scholar*. The queries for the literature reviews are "fuzz testing", "fuzzing", "fuzzer", "random testing", or "swarm testing" as the keywords of the titles. Figure 2.1 presents the results of the mentioned study.

A *run* is a sequence of instructions that connects the start and termination of a program. A successful run (execution) behaves as the program is intended to

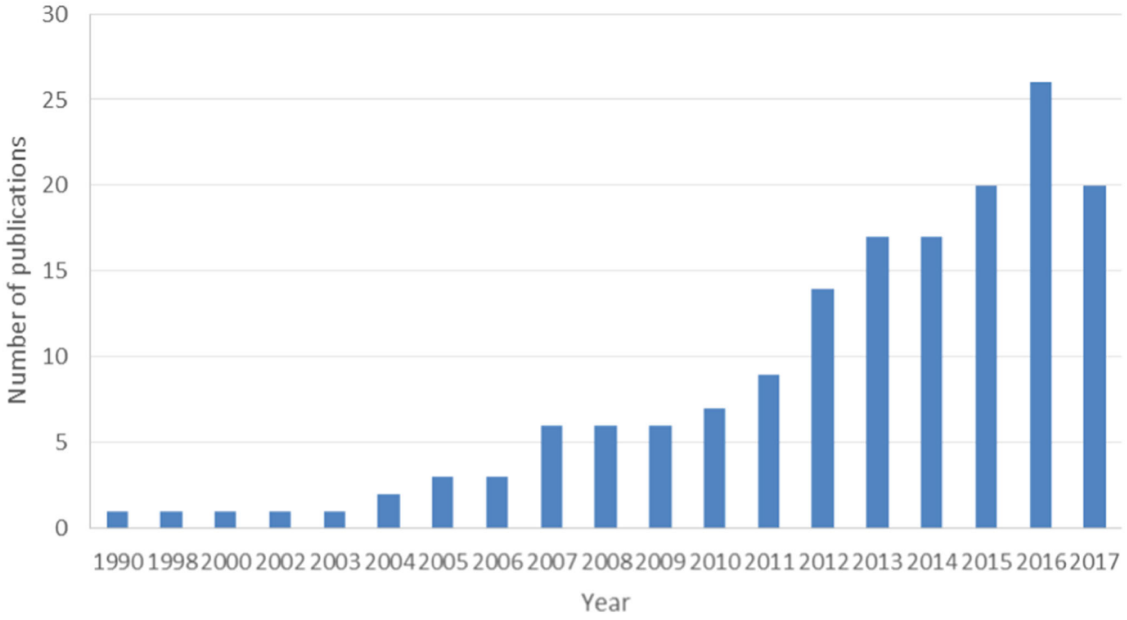


Figure 2.1: Fuzzing papers published between January 1st, 1990 and June 30 2017. [1]

run. An exception is a signal that is thrown, indicating an unexpected behavior. If the exception is not caught before the program's termination, the operating system receives an unfinished task with the exception's descriptions. From the OS's perspective, the program has crashed and could not finish its execution properly. A software vulnerability is an unexpected state of the program that is failed to be handled. Different states of a program occur by different inputs that a program takes. The inputs such as *environment variables*, *file paths*, or other program's *arguments* are mainly selected to search for the vulnerabilities.

Fuzz testing is the repetitive executions of a target program with different inputs. Fuzz testing takes two main actions in the fuzzing procedure: the fuzzer *generates test cases* for the target program, and each generated (fuzzed) test case is then passed to the program for *execution*. A fuzzer gathers information out of each execution. A *whitebox* fuzzer has access to the source code of the target program. *Analyzing the source code*, *monitoring the execution*, and *validating the returned value of execution*, are of the capabilities of a whitebox fuzzer. Oppositely, a *blackbox* fuzzer does not

have any access to the source code, cannot analyze the execution and does not check the result of the execution. Instead, a blackbox fuzzer focuses on executing more instances of the program blindly. Fuzzers with at least one property from each of the whitebox and blackbox fuzzers are in the category of *greybox* fuzzers.

The common strategies for fuzzing new test cases include *genetic algorithms*, *coverage-based (coverage-guided) strategies*, *performance fuzzing*, *symbolic execution*, *taint-based analysis*, etc. Genetic algorithms (GA) are *evolutionary algorithms* for generating solutions to *search* and *optimization problems*. GA has a population of solutions that their evaluations affect their survivability for the next generation. Inspired by the biological operations, GA processes the selected (survived) population and applies *mutations* and other modifications on them, resulting in a new generation of the population [6, 7]. Coverage-guided strategy is a genetic algorithm that utilizes *concrete analysis* of the *execution-path* of a program. A concrete analysis investigates the runtime information of an executive program, and the graph of the executed instructions (execution-path) can be collected through this analysis. Symbolic executions determine the constraints that change the execution-paths [8]. Performance fuzzing is a coverage-based technique that generates *pathological inputs*. "Pathological inputs are those inputs which exhibit worst-case algorithmic complexity in different components of the program" [9]. A taint-based analysis of a program tracks back the variables that cause a state of the executing program. This approach can detect vulnerabilities with no false positives [10].

American Fuzzy Lopper (AFL) [11] is a coverage-based greybox fuzzer, that is originally considering the number of times each *basic block* of execution is visited. Each basic block is a sequence of instructions with no branches except the entry (jump in) and exit (jump out) of the sequence. AFL is published with two default tools for collecting the runtime information: *LLVM* [12] and *QEMU* [13]. "The

LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project." AFL acts as a **whitebox** fuzzer in *llvm-mode*. In **llvm-mode**, AFL provides the recipe for compiling the target program with coverage information. The resulting compiler adds *static instrumentations* (**SI**) to the program. Instrumentation is the process of injecting logging instructions into the program, and SI refers to the instrumentations applied on a binary before execution. The added instructions store the code coverage and AFL can use them in Fuzzing. QEMU (Quick EMUlator) is an open-source emulator and virtualizer that helps AFL with *dynamic instrumentation* (**DI**). In DI, the instructions are inserted in runtime, and an emulator such as QEMU helps AFL with discovering the code coverage [14]. *DynamoRIO* [15], *Frida* [16], and *PIN* [17] are some other examples of pioneer DI tools.

In this chapter, we begin with reviewing the previous works that lead to this thesis 2.2. Next, we describe the implementation of AFL and its llvm-mode in **??**. We wrap up this chapter with conclusions.

2.2 Literature Review

Fuzzing searches for software vulnerabilities. "Vulnerability" has different definitions under various organizations and researches. For instance, *International Organization for Standardization (ISO)* defines vulnerability as: "A weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations and their continuity, including information resources that support the organization's mission." [18] Yet, the definition needs more details for software.

2.2.1 Software vulnerability

According to the *Open Web Application Security Project (OWASP)*: "A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows attackers cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application." The existence of software vulnerabilities may be compromised and may become an attack target for hackers; this makes the software unreliable for its users.

To exploit a vulnerability, an attacker calls an execution containing the bug and redirects the program's flow with appropriate inputs. An exploitable vulnerability may escalate privileges, leak information, modify/destroy protected data, stop services, execute malicious code, etc. [19]. An analysis of the program may prevent the exposure of vulnerabilities and stop further harm. Various techniques are used to identify weaknesses and vulnerabilities of software. Techniques such as *static analysis*, fuzzing, taint analysis and symbolic execution, etc. are the most common techniques that can be used cooperatively for error detection [20]. The static analysis evaluates the source code or binary to expose the vulnerabilities without executing the program.

To investigate a program for bugs, a model that helps the research is **Control Flow Graph (CFG)**. CFG is a directed graph whose nodes are the basic blocks of the program, and its edges are the flow path of the execution between two consecutive basic blocks. For instance, the Figure 2.2a illustrates the CFG for *bubblesort* algorithm (Algorithm 1). The branches in CFG split after a *conditional instruction* and a relevant *jump instruction*.

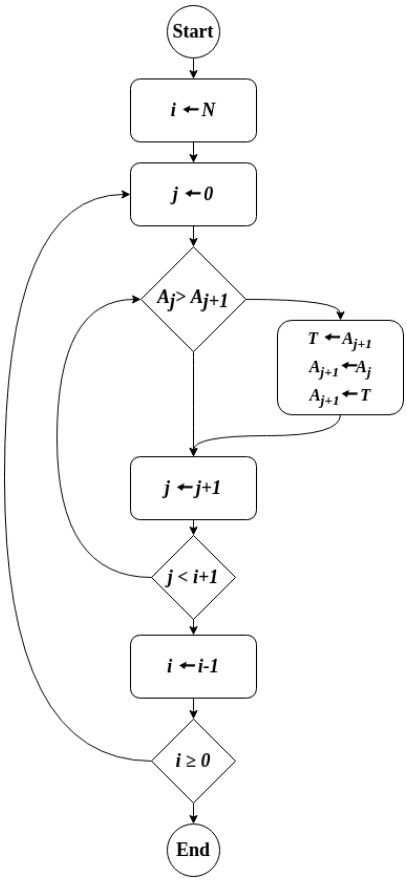
An execution processes a path (sequence) of instructions from an entry to any

Algorithm 1: Pseudocode of bubblesort on array A of size N

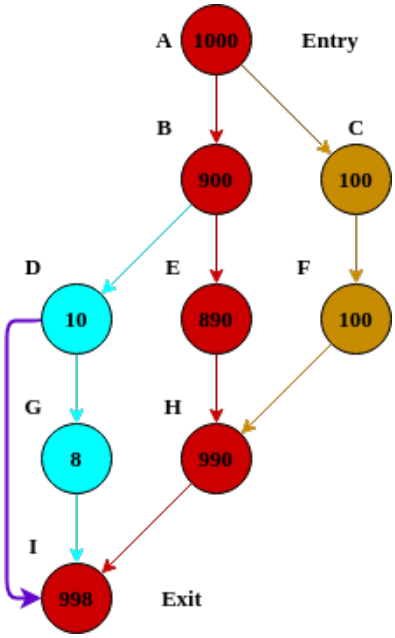
```
   Input: A, N
 1 i ← N;
 2 do
 3 |   j ← 0;
 4 |   do
 5 |   |   if A_j > A_{j+1} then
 6 |   |   |   SWAP(A_j, A_{j+1});
 7 |   |   j ← j + 1;
 8 |   while j < i + 1;
 9 |   i ← i − 1;
10 while i >= 0;
```

exit location of the program. For instance, consider Figure 2.2b as a CFG illustrating the executed paths of 1000 trials of the program's execution. The numbers in the basic blocks indicate the number of times each basic block is visited. A path such as $A \rightarrow B \rightarrow E \rightarrow H \rightarrow I$ has been explored more than other execution paths. Basic block D is visited occasionally and the edge $D \rightarrow I$ directly goes to the Exit, representing bugs in basic block D. These 1000 trials have discovered 9 seperable basic blocks, but it does not imply that there is no other basic blocks or edges revealed after more trials. Code coverage measures number of basic blocks which could be reached in an experiment of trials.

Denial of service (DoS) is a category of vulnerabilities through network that prevents services from correctly responding back to the users. "There are many ways to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses" [21]. In software domain, the vulnerability is either due to an early termination through a crash, or the program terminates with a timeout.



(a) Control Flow Graph of bubble-sort algorithm [Algorithm 1]



(b) CFG Heatmap of Algorithm 1. In 1000 runs, different paths are discovered.

Figure 2.2: Control Flow Graph

The vulnerabilities arise after target program executes with a triggering input. Miller introduced fuzz testing to examine the vulnerabilities of a collection of Unix utilities [22]. The results showed that a random fuzzing on different versions of the utilities could discover bugs in 28% of the targets. The automation in testing programs helps the researchers with validating the reliability of a program. The early fuzzers mimic a procedure of searching for the bugs by starting with *identification* of the target program and its inputs. Next, the fuzzing loop initiates, and the program is run with fuzzed inputs as long as the fuzz testing is not terminated. Figure 2.3 depicts the fuzz testing procedure defined by Sutton et al. [2]. Based on the definitions, a standard fuzzer consists of:

- Target identification
- Inputs identification
- Fuzzed data generation
- Execution of target with fuzzed data
- Exceptions monitoring
- Exploitability determination

Target is a software or a combination of executables and hardware [23]. A targeted software is any program that a machine can execute. Fuzzer needs to know the command for executing the target program and the inputs (arguments) of the program. **Inputs** are a set of environmental variables, file formats, and any other parameters that affect the execution. The initial seeds of the inputs can guide the fuzzer for finding more complex test cases, yet, it is not mandatory to provide seeds, and a fuzzer can generate valid inputs *out of thin air* [24]. After the initial setup, the **fuzzing loop** begins iterating. In each iteration the fuzzer **executes** the target with the **provided test cases**. Fuzzer then proceeds to detect exceptions returned

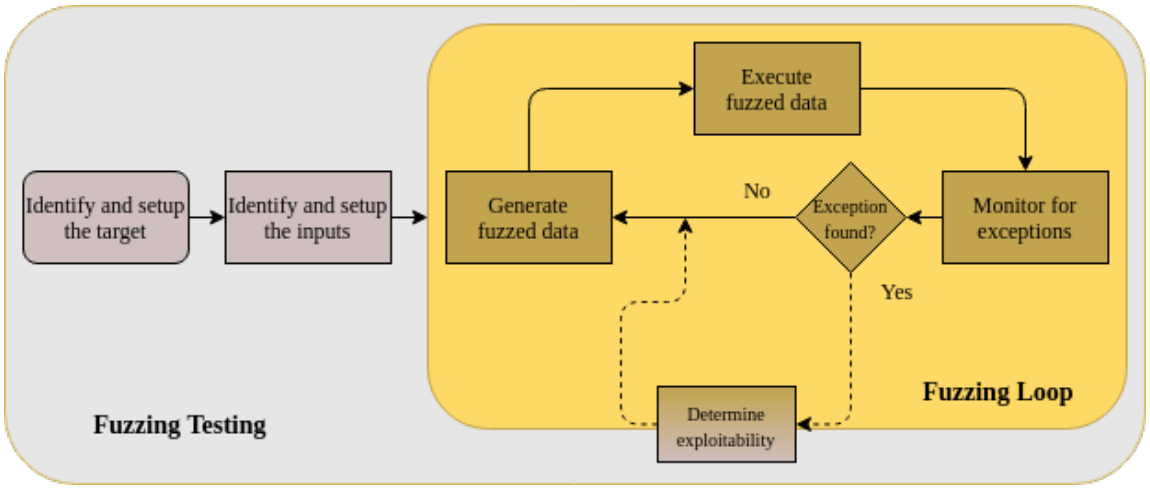


Figure 2.3: Fuzzing phases. Inspired by the definition of Sutton et al. [2]

from the executions, and considers the executed input responsible for causing a **vulnerability**. The vulnerability can then be analyzed for **exploitability** in the last stage. An exploitable vulnerability can compromise the system and initiate an anomaly.

The categories for fuzzers with different **program awareness** and different **techniques for fuzzing** the inputs help the community find different applications of the fuzzers for discovering more vulnerabilities. For instance, a developer uses a whitebox fuzzer to assess the immunity of the program (source-code accessible) against malicious activities. On the other hand, an attacker may use a blackbox fuzzer to attack a remote program blindly. A researcher may use a coverage-based fuzzer to consider the execution paths as a variable to reach more regions of the code and detect more crashes; hence, another researcher may use a performance fuzzer to reveal the test cases causing performance issues.

2.2.2 Program awareness

The *colorful* representation of fuzzers depends on the amount of information collected from a symbolic/concrete execution. A blackbox fuzzer does not gather any information from the execution. In contrast, whitebox fuzzers have all the required access to the program's source code, and greybox fuzzing covers the gray area between the mentioned types.

2.2.2.1 Blackbox fuzzing

Blackbox fuzz testing is a general method of testing an application without struggling with the analysis of the program itself. The target of an analysis executes after calling the proper API's, and the errors are expected to occur in the procedure of trying

various inputs. Blackbox fuzzing is an effective technique though its simplicity [25].

The introduced fuzzer by Miller [22] was of the very first naive blackbox fuzzers. It runs the fuzzing for different lengths of inputs for each target (of the total 88 Unix utilities) and expects a **crash**, **hang**, or a **succeed** after the execution of the program. Each input is then fuzzed with a random mutation to generate new test cases. One of the **downsides** of blackbox fuzzing is that the program may face branches with *magic values*, constraining the variables to a specific set of values; for instance, as shown in Listing 2.1, the chance of satisfying the equation `magic_string=="M4G!C"` and taking the `succeed()` path is almost zero. In [26] and [27] a set of network protocols are fuzzed in a blackbox manner, but as the target is specified, the performance is enhanced drastically. Any application on the web may be considered a blackboxed program as well, so as [28] and [29] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

```
1  string magic_string = random_string();
2  if(magic_string == "M4G!C")
3      return succeed();
4  else
5      return failed();
```

Listing 2.1: Magic Value: `M4G!C` is a magic value

A blackbox fuzzer is unaware of the program's structure and cannot monitor it's execution. The **benefit** of using a blackbox fuzzer is the speed of test case generation; the genuine compiled target program is being tested and the fuzzer does not put an effort on processing the inputs and executions. In addition, a blackbox fuzzer is featured to target external programs by using the standard interfaces of those programs. For instance, IoTFuzzer [30] is an Internet of Things (IOT) blackbox fuzzer, "which aims at finding memory corruption vulnerabilities in IoT devices without access to their firmware images." In a recent research by Mansur et al. [31], they introduce a blackbox fuzzing method for detecting bugs in Satisfiability Modulo

Theories (SMT) problems. As a result, blackbox fuzzing suggests a general solution in diverse domains. On the other hand, one of **drawbacks** of using blackbox fuzzing is that it finds *shallow* bugs. A shallow vulnerability is an error that appears in the early discovered basic blocks in the CFG of the program. The reason behind this disadvantage is that blackbox fuzzing is **blind** in understanding the execution, and cannot analyze the CFG.

2.2.2.2 Whitebox fuzzing

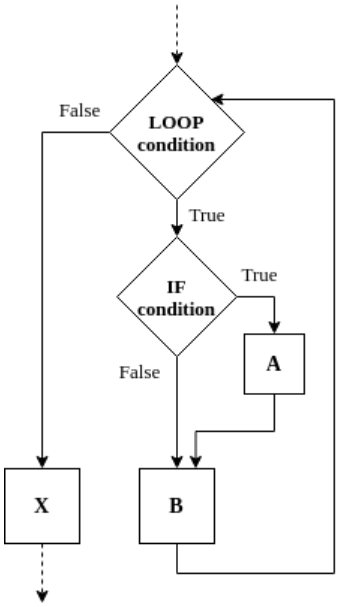
Whitebox fuzzing works with the source code of the target. The source code contains the logic of the program and can anticipate the executions' behavior without executing the program (concretely). Symbolic execution [32] is a reliable whitebox fuzzing strategy that analyzes the source code. This analysis replaces the variables with symbols that consider the constraints for each data. This technique helps its fuzzer discover inputs that increase code coverage by discovering new branches after conditional instructions were satisfied. This method detects *hidden* bugs faster due to the powerful constraint solvers [33]. Although the symbolic execution can solve the conditional branching theoretically, this technique suffers from *path explosion* problem. As an example, in Figure 2.4a a sample section of a program containing a `loop` and an `if` statement within the loop. Figure 2.4b shows the tree of the actions taken until the program reaches the basic block X. Solving the current loop requires an exponentially growing number of paths that the fuzzer needs to visit. Whitebox fuzzing is not very practical in the industry as it is expensive (time-consuming / resource-consuming) and requires the source code, which may not be available for testers.

SAGE [34], a whitebox fuzzer, was developed as an alternative to blackbox fuzzing to cover the lack of blackbox fuzzers [35]. It can also use dynamic and *concolic*

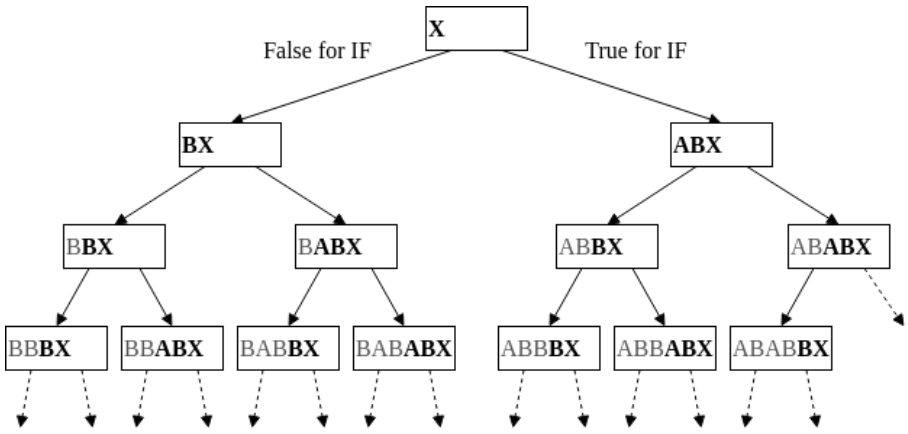
execution [36] and use taint analysis to locate the regions of seed files influencing values used by the program [37]. Concolic execution is an effective combination of symbolic execution and concrete (dynamic) executions; in a dynamic execution the fuzzer executes the program and analyzes the run. Godefroid et al. [38] have also introduced a whitebox fuzzer that investigates the grammar for parsing the input files without any prior knowledge.

2.2.2.3 Greybox fuzzing

Greybox fuzzing resides between whitebox and blackbox fuzzing, as it has partial knowledge (awareness) about the internals of the target application. The source code is not analyzed, but the executions of the binary files are the main data source for discovering the vulnerabilities in action; the actual application's logic is not considered for the analysis, but the instructions illustrate an overview of the compiled



(a) Path explosion



(b) Path explosion tree

Figure 2.4: Path explosion example

program's logic. A concrete execution of a program represents a reproducible procedure that is executed and can be monitored for its behavior detection. Greybox fuzzer obtains the runtime information from the code instrumentation, and by using other techniques such as taint analysis, concolic executions (obtaining the logic from the binary), and methods for acquiring more information after the **partial knowledge** of the program [1, 39].

The code coverage is a viable feature for detecting the new paths that the fuzzer never executed. Registering new paths for testing helps fuzzers in finding different regions of the code for potential vulnerabilities. Later the fuzzer tests the input that caused the new path to check if fuzzing the input can reveal a new vulnerability after constructing tweaked inputs out of the current input. This loop of testing the different inputs continues until specific termination signals show up. AFLGo [40] is a greybox fuzzer that tries digging into the deeper application's basic blocks so that it can reach a specific region. AFLGo focuses on fuzzing more of the inputs that guide the execution as it gets closer to the specified region. This greybox fuzzer shows effective performance in testing a program for detecting errors in new patches of an application and helps with crash reproduction. Another greybox fuzzer, VUzzer [41] enhances the instrumentation to collect control- and data-flow features, which leads to guiding the agnostic fuzzer to find more *intersting* inputs, with less effort (fewer trials of the fuzzed inputs). The core feature in a greybox fuzzer is the *application agnosic* characteristic that targets any executable and observable program in the fuzzer's environment.

2.2.3 Input generation

Greybox fuzz testing requires features to distinguish between various inputs and pick the test cases that help the fuzzer find bugs. Code coverage is a distinguishing feature

for preference over the inputs. As described before 2.2.1, code coverage summarizes the behavior of an execution, and does not analyze the instructions, instead, the graph of the basic blocks (CFG) is analyzed. For **coverage-guided** fuzzing, the corpus of inputs extends as the fuzzer finds new *execution paths*. On the other hand, a **performance-guided** fuzzer seeks *resource-exhaustive procedures*.

2.2.3.1 Coverage-guided fuzzing

Coverage-based fuzzing is a technique for fuzz testing that instruments the target without analyzing the logic of the program. In a greybox and whitebox coverage-based fuzzing, the instrumentation detects the different paths of the executions [1]. The applied instrumentation collects runtime information such as data coverage, statement coverage, block coverage, decision coverage, and path coverage [42]. Bohme et al. [8] introduced a coverage-based greybox fuzzer that benefits from the Markov Chain model. The fuzzer calculates the *energy* of the inputs based on the **potency of a path for discovery of new paths**. Later, the inputs with higher energy add more fuzzed inputs to the queue.

Steelix [43] is a coverage-guided greybox fuzzer. It implements a coverage-based fuzz testing that is boosted with a *program-state based* instrumentation for collecting the *comparison progress* of the program. The comparison progress keeps the information about *interesting comparisons*. The heavy-weight fuzzing process of Steelix contains an initial light-weight static analysis of the binary. The static analysis returns the basic block and comparison information. Later, the concrete execution of the fuzzed test cases determines the state of the program based on the triggered comparison jumps. In addition to the coverage increasing generation of inputs, Steelix knows how to solve the *magic* comparisons, and looks for new states of the program based on the resolved comparisons.

Types	Benefits	Limitations	Example Fuzzers
Whitebox	Deep/Hidden bug finding	- Path Explosion - Accessibility to source code	SAGE BuzzFuzz
Blackbox	- Fast test case generation - General applicability	- Shallow bug finding - Blind	LigRE Storm
Greybox	- Deep bug finding - General applicability	- Access to binary - Requires instrumentation	AFL LibFuzzer

Table 2.1: Program awareness for fuzzing

The code coverage is measured by considering a light-weight instrumentations. This method helps fuzzing to monitor the program without changing the program's resource usage (time, memory, etc.) by a noticeable amount. Hence, due to unaccessibility to the source code, dynamic instrumentation is used as a reliable technique for collecting the runtime information. The trade off for using DI is the increasing performance cost; for instance, QEMU costs 2-5x slower executions [14].

2.2.3.2 Performance-guided fuzzing

To enhance the capability of a coverage-based fuzzer, a performance-guided fuzzing technique collects resource-usage information, and leverages code coverage techniques for exploring the CFG. SlowFuzz [44] is a performance-guided coverage-based greybox fuzzer, which measures the length of the executed instructions in a total (complete) execution. SlowFuzz has an interest in evolving the corpus of test cases to discover new paths or generate more resource exhaustive executions. Another

Types	Features	Drawbacks	Examples
Coverage-guided	Light-weight instrumentation	Low guidance measurements	AFL LibFuzzer Honggfuzz
Performance-guided	- More features for guidance - Find resource-exhaustion	Heavy weight instrumentation	SlowFuzz PerfFuzz MemLock

Table 2.2: Input generation techniques for fuzzing

performance-guided fuzzer, PerfFuzz [9] aims to generate inputs for executions with higher **execution time** by counting the number of times each edge (jump out of basic block) of the CFG is visited. Next, inputs with higher edge counts take more effect on the corpora's evolution. These two performance fuzzers can detect pathological inputs related to CPU usage. Memlock [45] guides the performance of the fuzzing to produce memory-exhaustive inputs. It investigates memory usage by calculating the maximum runtime memory required during executions. MemLock uses static performance instrumentations for profiling memory usage.

2.3 American Fuzzy Lopper (AFL)

Michal Zalewski developed American Fuzzy Lopper as a coverage-guided greybox fuzzer. He introduces this open-source project as "a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test inputs that trigger new internal states of the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpus produced by the tool help seed other, more labor- or resource-intensive testing regimes down the road." [46] AFL is designed to perform **fast** and **reliable**, and at the same time, benefits from the **simplicity** and **chainability** features [47]:

- **Speed:** Avoiding the time-consuming operations and increasing the number of executions over time.
- **Reliability:** AFL takes strategies that are program-agnostic, leveraging only the coverage metrics for more discoveries. This feature helps the fuzzer to perform consistently in finding the vulnerabilities in different programs.

- **Simplicity:** AFL provides different options, helping the users enhance the fuzz testing in a straightforward and meaningful way.
- **Chainability:** AFL can test any binary which is executable and is not constrained by the target software. A driver for the target program can connect the binary to the fuzzer.

AFL tests the program by running the program and monitoring the execution path for each run. To extract information from a run, AFL offers multiple **instrumentation** techniques for constructing hashes of the explored paths while following the executing path of the actual program. AFL requires instrumented binaries which provide the execution information when they are run by AFL. AFL is a whitebox/greybox fuzzer when it can effectively insert the instrumentations into the program. Figure 2.5 shows a simplified illustration of the procedure of AFL. In whitebox fuzzing, AFL takes the source code of the program and executes static instrumentation during the compilation of the program, and passes the generated binary to the fuzzing module. On the other hand, the greybox feature of AFL lets the fuzzer to execute the un-instrumented binary under a dynamic instrumentation,

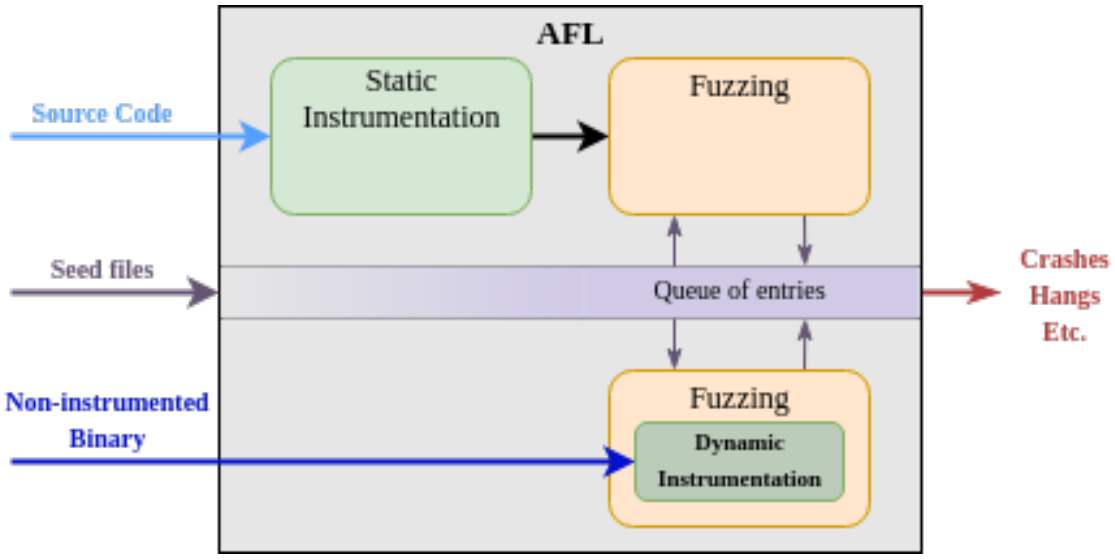


Figure 2.5: AFL's procedure: simplified

which executes the instrumenting instructions wrapped around the basic blocks of the program.

2.3.1 Instrumentation

The collection of coverage information is generated during the execution; when the execution steps into a basicblock, the procedure 2.2 stores the visited edge (pair of two consecutive basicblocks) in CFG. The hashing only stores the information from the previous basicblock and the current basicblock which we are already in. For instance, suppose we have CFG of a program as shown in 2.6; by giving the persmission to the program to modify a memory region shared with AFL - which AFL also has access to it - the coverage instructions generate a summary of the executed path. For example, suppose we have an instrumented program with the random values which are set in compile time (for simplicity, suppose that initially random value $cur\_location = 1010$). Running the first basic block assigns CUR_HASH to $73 \oplus 1010 = 955$; the content of index 955 of shared memory is then increased by one, and the execution continues to the next basic block. If the execution is jumped into basicblock 2, the content of $shared\_mem[310 \oplus (73 >> 1) = 274]$ is incremented by one, and so on. This procedure continues along with the main execution and in the end, the content of the shared memory contains a hashing of the traveled path. In this scenario, taking the path $1 \rightarrow 2 \rightarrow 5$ results in an array of zeros except for $\{51 : 1, 274 : 1, 955 : 1\}$ as the hashed path.

```
1    cur_location = <COMPILE_TIME_RANDOM>;
2    shared_mem[cur_location ^ prev_location]++;
3    prev_location = cur_location >> 1;
```

Listing 2.2: Select element and update in shared_mem

AFL uses both dynamic and static instrumentation for profiling executions. Static instrumentation is applied using **LLVM** modules which can analyze and insert

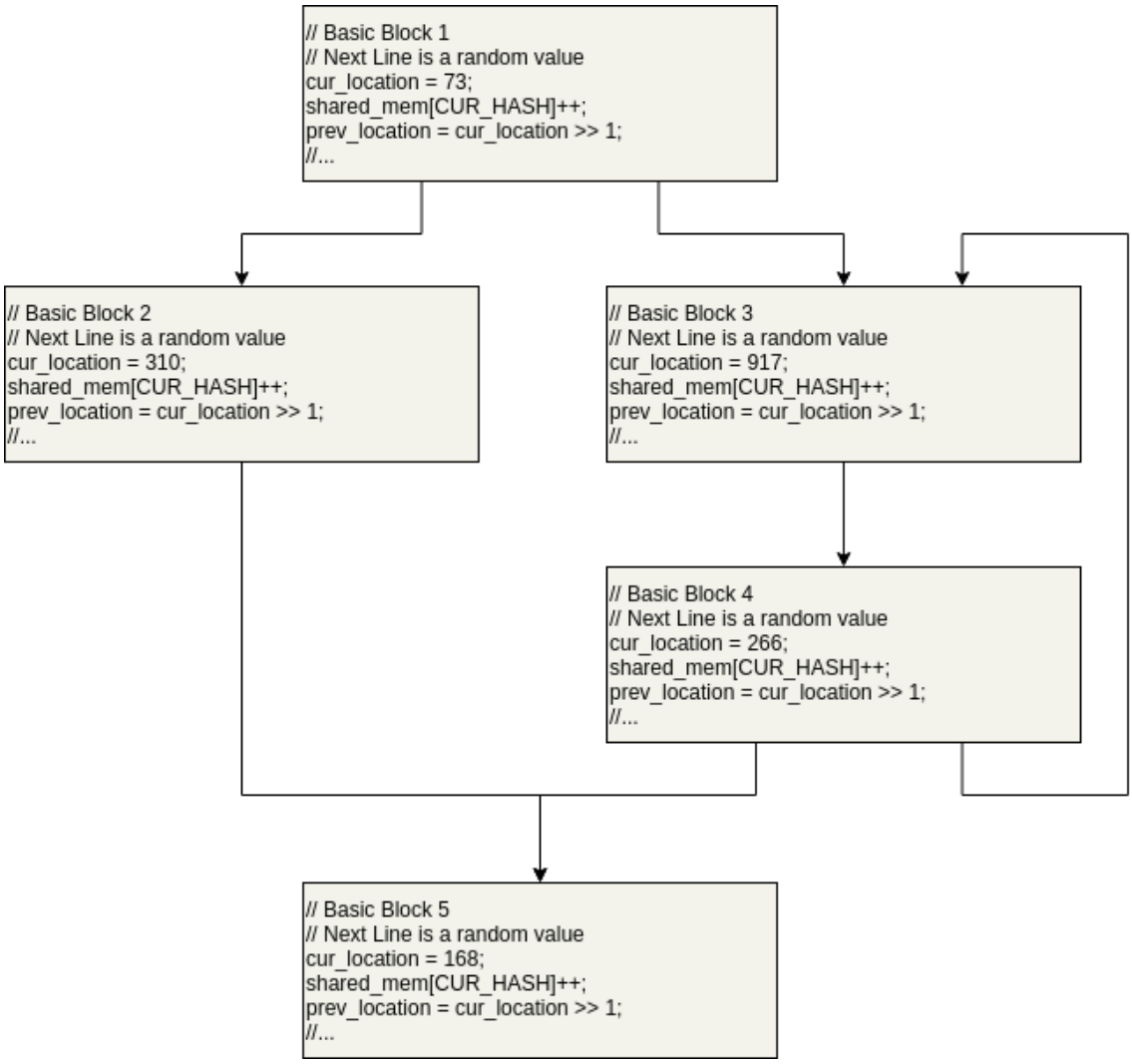


Figure 2.6: Example for instrumented basic blocks

instructions anywhere in the program. Dynamic instrumentation is another method which let let the fuzzer to use the instrumentation instructions while executing the program. By default, AFL uses `QEMU` for dynamic instrumentation [14]. The emulator wraps the genuine instructions into analyzable modules, and constructs the execution path while running. This technique, **qemu-mode**, causes a slow down of 10-100% for each execution compared to the **llvm-mode**. For the purpose of this article, we need to dig more into the procedure of instrumentation in llvm-mode [48].

2.3.1.1 LLVM

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name **LLVM** itself is not an acronym; it is the full name of the project." [12] Two of the relevant projects used by AFL are:

- The **LLVM Core** libraries contain source/target-independent optimizers as well as code generators for popular CPUs. These well-documented modules assist development of a custom compiler in every step (*pass*) through the conversion of source code to executable binary file.
- **Clang** [49] provides a front-end for compiling C language family (C, C++,

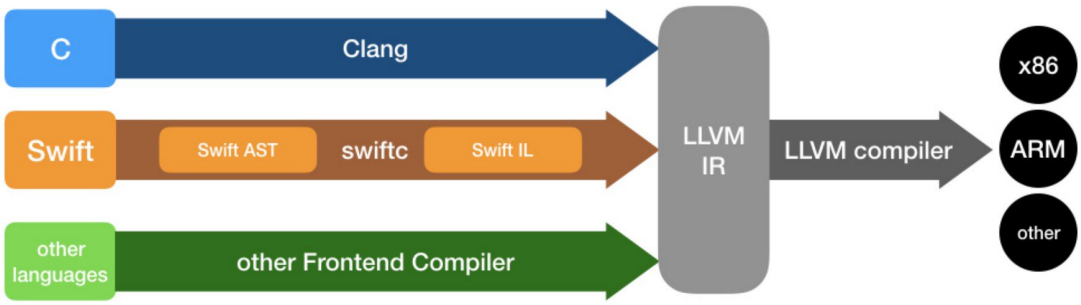


Figure 2.7: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [3]

Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM Project. Clang uses the LLVM Core libraries to generate an *Intermediate Representation* (**IR**) of the source code [50]. The IR is then translated into an executable binary for the machine's CPU (Figure 2.7).

AFL utilizes the compilation *passes* of Clang with a custom recipe for *module pass*. Passes are the modules performing the transformations and optimizations of a compilation. Each pass is applied on a specified section of the code. For instance, the **ModulePass** class (and any classes derived from this class) performs the analysis of the code and the insertion of new instructions. Other classes such as **FunctionPass**, **LoopPass**, and etc perform their instructions using different parts of the code, and they contain less information about the rest of the program. AFL uses only the ModulePass and iterates over every basicblock existing in the program

2.3.2 AFL Fuzz

The automatic fashion of testing a targeted software using AFL requires a modified execution of the software with coverage-based instructions. The coverage-based instructions are used during the fuzz testing to provide a summary of the execution under AFL's supervision. AFL passes the next test case from the queue of entries to the program, and *caliberates* the test case so that it is ready to be fuzzed. After the calibration of the case, AFL tries to trim the case such that the execution's profile remains the same.

AFL fuzzes the case after it's preperations; different mutation techniques are applied to the test case, and the newly generated test cases are analyzed to check if they are producing any new execution behavior based on their code coverage and execution's $speed \times file\_size$. The cases which pass the prior check are considered

as *interesting* cases and are added to the queue of entries. We will investigate the details of this functionality later in this article.

AFL uses various mutation techniques for fuzzing a case. There are two main stages for mutations:

- **Deterministic** stage: This early stage contains a sequence of operations which contain:
  1. Sequential bit flips with varying lengths and stepovers
  2. Sequential additions and subtractions of small integers
  3. Sequential insertion of know interesting integers - such as *0*, *1*, *MAX_INT* and etc. [47]
  4. Sequential replacement of content of the case by dictionary tokens provided: AFL accepts a dictionary of known values - such as magic values used in the header of a file. For instance, a dictionary for HTML tags which is also provided in the AFL's repository, contains known HTML patterns such as `tag_header="<header>"` , which AFL would use to replace some bytes of the content of the case with string `<header>`.
- **Non-deterministic** stage: This stage contains two main operations:
  1. Random HAVOC: A sequence of random mutations are applied on the input in this stage. The operations include actions on file such as insertion and deletion of random bytes, cloning subsequences of input, and etc. The repetition of this stage depends on the **performance score** of the current case under investigation. AFL calculates this score based on the code coverage of the test case and it's execution speed; the more locations visited stored in *coverage bitmap*, and the lower the execution speed are prefereable, and as a result, AFL increases the score based on that.

2. Splicing: If non of the previous stages result in any new findings, AFL tries selecting a random case from the queue, and copies a subsequence of that case into the current case, and relies on the HAVOC stage for mutating the results.

As we discussed, the input files generated for the target program after the fuzzing phase are processed for any *interesting* feature. This procedure goes through two functions called as `save_if_interesting()` and *update_bitmap_score()*:

- `save_if_interesting()`: This function checks if there are any new bits in the shared trace bits (which stores the coverage information) and if a finding is showing new previously unset bits, it tags the case as an interesting one. This function also summerizes the trace bits to reduce the processing effort in later checks - to compare new findings with this generated case.
- `update_bitmap_score()`: AFL maintains a list of `top_rated[]` entries for every byte in the bitmap. Each element of the `top_rated[]` entries tracks the fastest `queue_entry` which visits an edge.

AFL analyzes the cases using the above provided functions, and the best queue entries are updated eventually in the fuzzing procedure. To select the next element of the queue for fuzzing, AFL uses this information for tagging the *favorite* and *redundant* entries. These operations are done under the function *cull_queue()*. Each element of the list `top_rated[]` contains a pointer to the `queue_entries[]`, and marks the pointed entry as a `favored` entry. After a complete walk over the list, the remaining unfavored entries are marked as `redundant`. In every fuzzing cycle over the queue entries, `cull_queue()` prepares the queue and then AFL takes the next front entry which is also *favored*.

2.3.3 Status screen

The **status screen** is a UI for the status of the fuzzing procedure. As it is shown in Figure 2.8, there are various stats provided in real-time updates:

1. **Process timing**: This section tells about how long the fuzzing process is running.

2. **Overall results**: A simplified information about the progress of AFL in finding paths, hangs, and crashes.

3. **Cycle progress**: As mentioned before, AFL takes one input and repeats mutating it for a while. This section shows the information about the current cycle that the fuzzer is working with.

4. **Map coverage**: The AFL's documentation explains the information in this section as: "The section provides some trivia about the coverage observed by the instrumentation embedded in the target binary. The first line in the box tells you how many branches we have already hit, in proportion to how much

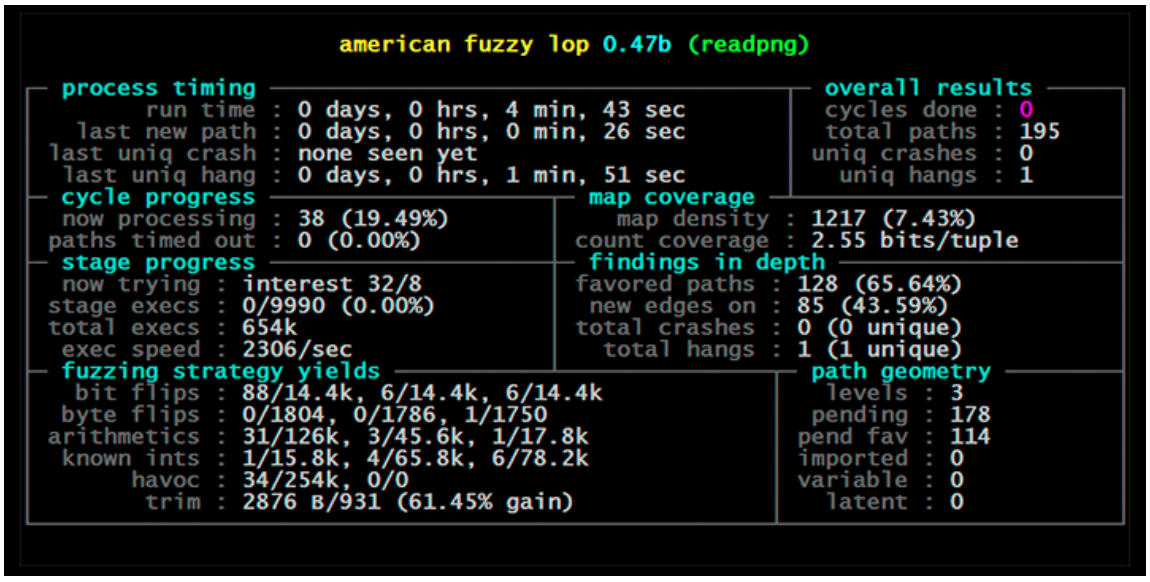


Figure 2.8: AFL status screen

the bitmap can hold. The number on the left describes the current input; the one on the right is the entire input corpus's value. The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every branch, the needle will start to move toward "8.00" (every bit in the 8-bit map hit) but will probably never reach that extreme.

Together, the values can help compare the coverage of several different fuzzing jobs that rely on the same instrumented binary."

5. **Stage progress**: The information about the current mutation stage is briefly provided here. This regards to `fuzz_one()` function in which the new fuzzed inputs are being generated through mutation stages.

6. **Findings in depth**: The number of crashes and hangs and any other findings are presented in this section.

7. **Fuzzing strategy yields**: To illustrate more stats about the strategies used since the beginning of fuzzing. For the comparison of those strategies, AFL keeps track of how many paths it has explored, in proportion to the number of executions attempted.

8. **Path geometry**: The information about the inputs and their depths, which says how many generations of different paths were produced in the process. The depth of an input referes to which generation the input belongs to. Considering the first input seeds as depth 0, the generated population from these inputs increase the depth by one. This shows how far the fuzzing has progressed.

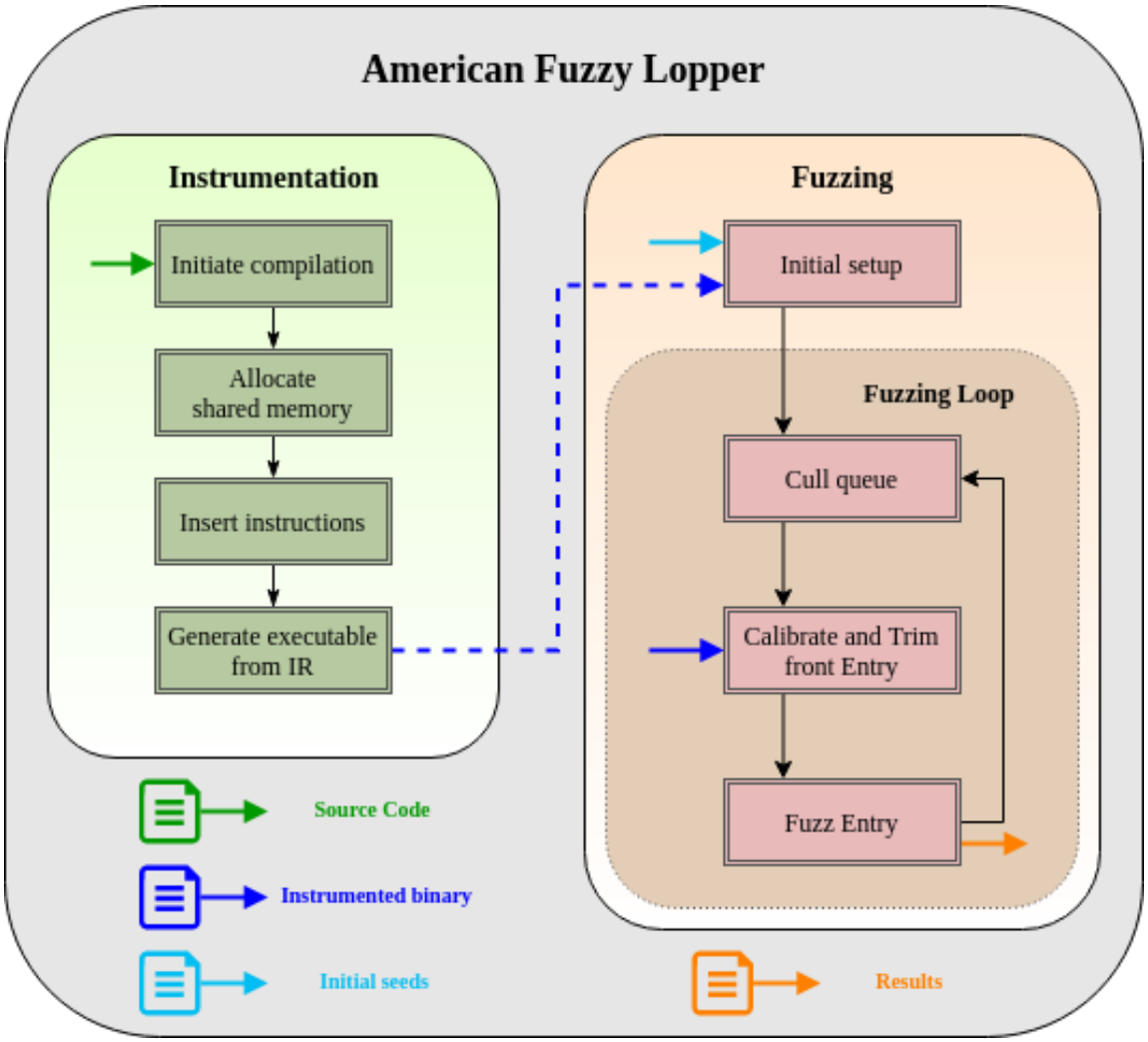


Figure 2.9: An overview of the whole fuzzing procedure of AFL

2.3.4 AFL fuzzing chain

Figure 2.9 illustrates the procedure of fuzzing, from static intrumentation to fuzz testing the target and outputing results. To start the procedure, AFL instrumentates the program and configures the target binary for fuzz testing stage. To perform the instrumentation, AFL calls its own compiler, which applies the instrumentation when it's process is finished [Listing 2.3]:

```
afl-clang sample.c -o sample_inst
```

Listing 2.3: Instrument *sample_vul*.c

The result file, `sample_inst`, is an executable which contains shared memory for later analysis in fuzzing. Now AFL can start testing the program for probable vulnerabilities. The test requires the input and output directories, as well as the command for executing the program [Listing 2.4]. The fuzzing continues until recieving a halt signal (For instance, by pressing *Ctrl+C*).

```
# afl-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
afl-fuzz -i in_dir -o out_dir -- ./sample_inst
```

Listing 2.4: Execute AFL