École Polytechnique Fédérale de Lausanne

Regression Use-After-Free Precondition Fuzzing

by Erchang Ni

Master Project Report

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Gwangmu Lee
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 24, 2024

Abstract

In this study, we introduce a novel fuzzing approach known as "Regression Use-After-Free Precondition Fuzzing." This method focuses on identifying regression use-after-free vulnerabilities, a significant concern in nowadays software security.

By instrumenting basic blocks modified in recent commits or containing 'free' operations, we generate a precondition map during the execution of the target program. This map then guides the fuzzer, with inputs correlated to precondition edges being added to the seed queue. Inputs associated with a larger precondition map are allocated more fuzzing time. This technique also harnesses the power schedule and weighted sampling methods to give priority to inputs that execute code which has been changed more recently and frequently.

Our evaluation, conducted on a benchmark comprising seven use-after-free and double-free bugs in open-source C programs, demonstrates the enhanced efficiency of this method when compared to traditional fuzzing tools like AFL and AFLChurn.

Contents

Abstract (English/Français) **2**

1 Introduction **5**

2 Background **8**
2.1 Fuzzing . . . 8
2.1.1 Coverage-guided Fuzzing . . . 9
2.1.2 Regression Fuzzing . . . 9
2.2 Use-after-free . . . 10
2.2.1 Introduction . . . 10
2.2.2 Real-world Examples . . . 11

3 Design **13**
3.1 Precondition-guided Regression Fuzz Testing . . . 13
3.1.1 Workflow Explaination . . . 13
3.1.2 Scoring Mechanism . . . 15
3.2 Precondition Map Construction . . . 16
3.3 Commits-based Instrumentation Period Selection . . . 18

4 Implementation **19**
4.1 Source Code Compiler-level Instrumentation . . . 19
4.1.1 Instrumenting Projects with Different Scale . . . 20
4.1.2 Free-based Instrumentation . . . 21
4.2 Cyclic Queue and Precondition Map Size Selection . . . 22

5 Evaluation **23**
5.1 Research Questions . . . 23
5.2 Benchmark Subjects . . . 23
5.3 Setup and Infrastructure . . . 24
5.4 Experiment Results . . . 25

6 Related Work **27**

7 Conclusion **29**

Bibliography **30**

A Performance Score Augmented to AFL **33**

B Identify Changed Basic Blocks **34**

Chapter 1

Introduction

The estimated costs for system maintenance are believed to account for a minimum of 50% of the overall expenses throughout a system's lifecycle [21]. Research suggests that these maintenance expenses vary by industry, occupying about 49% of total costs in the pharmaceutical sector and escalating to 75% in the automotive industry [18]. Developers might publish new versions of codes and products to fix newly discovered bugs, improve performance, add new features, or adapt to new environments. For example, to fix discovered bugs, on average, the developers allocate about 45% to 80% of their working time and effort to identify, comprehend, and fix the bugs [2].

Nowadays, large software companies such as Google and Microsoft are spending a large amount of computational resources to perform grey-box fuzzing to discover security flaws in programs. For example, in Google's OSSFuzz project, a continuous fuzzing service, they employ three grey-box fuzzers: libFuzzer, AFL++, and Honggfuzz fuzzing engines in combination with Sanitizers, to uncover bugs in over 300 open-source projects [25]. Grey-box fuzzing is designed to randomly mutate inputs to discover vulnerabilities without specific knowledge about the program's internal structures, identifying potential vulnerabilities through the generation of a large scale of random test cases [8].

Among all the fuzzing technologies, American Fuzzy Lop (AFL) holds a pivotal place in the history of fuzzing. As one of the most influential fuzzing tools, it revolutionized the practice of automated software testing. AFL uses the Coverage-guided Fuzzing technology, Coverage-guided Greybox Fuzzing (CGF) is acknowledged as one of the most effective and successful techniques, which instruments target programs to track code execution, the fuzzer utilizes the edge information to prioritize mutations that trigger untested areas, thereby evolving inputs that can reveal bugs. Coverage-guided Grey-box Fuzzing operates on a genetic approach to search for better inputs that enhance and maximize code coverage.

However, even though those technologies are proven effective, because they treat the entire

codebase uniformly, it doesn't prioritize recent changes or the parts of the code that are modified often, where regression bugs are more likely to occur. Especially in those large and complex software projects, spending too much time on stable code in a huge project can dramatically decrease the effectiveness of traditional grey-box fuzzing. Regression bugs are prevalent in continuously evolving projects. Although developers implement software version updates out of good intentions, and those code updates are unavoidable, software changes inevitably introduce defects in the software. Those unintended behaviours introduced by source code updates are called regression bugs, they arise when a feature in the software that previously operated correctly ceases to function as intended or deviates from its expected performance. Research shows that back in 2007, 455 regression bugs were registered in the Apache Software Foundation bug tracking system [9], and an experimental study on OSSFuzz bug reports showed that shows that 77.2% of the 23k bugs reported are regressions, with an average of 68 days to discover and report after the bug is introduced [25].

To mitigate the negative impact and expense of regression bugs, the best practice is to consistently check the product's functionality after any updates to the source code. Continuously and comprehensively testing the source code plays a vital role in decreasing both the frequency and severity of regression bugs. Among diverse types of regression test technologies to automatically identify regression bugs, Regression Greybox Fuzzing (RGF) has stood out in recent years. To address the traditional greybox fuzzing technology such as AFL's limitation in searching regression vulnerabilities, RGF assigns higher priority to code segments based on their recent modifications and frequency of changes, factors denoted as *age* and *churn*. To effectively exploit the age and churn information, RGF computes a fitness value for each input during its execution. It employs a simulated annealing-based power schedule to steer the fuzzing towards inputs with higher fitness, and further utilizes ant colony optimization as byte-level power schedule to assign more energy to the most interesting bytes. The RGF has proven efficient in uncovering regression bugs and serves as a foundational element in our project.

However, the vanilla RGF overlooks the fact that most serious crashes only happen when their preconditions are met, such as free functions in use-after-free crashes. Since RGF indiscriminately prioritizes the inputs reaching patches, it wastes fuzzing time in the inputs that have not triggered the necessary preconditions, which are impossible to trigger crashes in the first place. Thus our project introduces a new method called regression precondition fuzzing, which augments regression greybox fuzzing by taking into account the precondition and effectively identifies regression *use-after-free* and *double-free* bugs in open-source C/C++ projects.

My main contributions are:

- **Technique**. We propose regression precondition fuzzing, a novel approach that constructs a precondition map by instrumenting basic blocks edited by source code's most recent commits. While executing the target program, the transition between *free* basic blocks and basic blocks in targeting commits will be tracked and reflected in the map.

- **Implementation**. We utilize constructed precondition maps, add inputs that lead to new precondition transitions to the queue for further testing, and combine with AFLChurn's power schedule, ACO search heuristic, and weighted sampling method, to effectively discover regression use-after-free vulnerabilities.
- **Evaluation**. we created a benchmark consisting of seven Use-after-free and Double-free bugs found in open-source C programs on GitHub. We conducted a comparative evaluation of our method against well-known fuzzing tools AFL and AFLChurn, focusing on their performance in identifying these bugs.

Section 2 provides background to contextualize our work. Section 3 introduces the design of our study, and Section 4 explains the ground-truth implementation of our design. Section 5 provides the evaluation results and explains them. Section 6 overviews related research and practices, and Section 7 concludes.

Chapter 2

Background

2.1 Fuzzing

Fuzz testing is a security testing technology that has gained high attention nowadays. In 1990, Barton Miller and his students carried out the first fuzzing test [12]. In 1995, they conducted UNIX utility fuzz testing for the first time. During the fuzz test, Miller's team found that nearly a third of the UNIX utility software crashed, including some with `setuid` rights that led to security vulnerabilities that could be exploited. In 2001, the PROTOS team tested, found, and fixed several severe vulnerabilities in the Internet's core protocol stacks, which were related to fuzz testing [14]. By 2002, Aitel's SPIKE designed a new framework that facilitated the development of new fuzzing tools [1]. In 2009, Charlie Miller's fuzzer won the Pwn2Own competition, demonstrating the practical value of fuzzing.

Fuzzing uses different technical means for various testing objectives, combining test case generation strategies to produce test cases. These cases are then fed into the target system, and potential vulnerabilities are identified by monitoring for abnormal results. Specifically, fuzz testing utilizes techniques such as instrumentation, symbolic execution, and taint analysis to gather information about the target. It employs various testing methods based on the degree of dependency on internal information of the target, and can be categorized into black-box, white-box, and grey-box testing. It relies on seed generation strategies to automatically or semi-automatically mutate normal data to produce a large number of expected or unexpected test cases. The method detects security vulnerabilities by analyzing the output results of the target. Fuzz testing has become the most important tool in the discovery of vulnerabilities in software and systems.

2.1.1 Coverage-guided Fuzzing

Our work, as a Coverage-guided Grey-box Fuzzing technology, is based on the idea that automatically generates test cases to execute unexplored areas of a program to find bugs [6]. Algorithm 1 outlines a coverage-guided fuzzing process. It starts with a set of initial seeds and executes the target program with these inputs to establish a baseline of code coverage. As the fuzzer runs, during each loop it selects inputs from the current input pool, applies random modifications, and monitors the program's execution to collect coverage data and check for new code paths and potential crashes or errors. When a mutation uncovers previously untested code, the fuzzer adds this input to its corpus for further mutation and testing. If an error is detected, it reports the issue. The tool employs a genetic algorithm approach, where test cases that lead to new coverage are favoured, essentially "evolving" the input set. In the real world, AFL will optimise this process using a power schedule that smartly allocates more energy to promising inputs, increasing the likelihood of discovering bugs in less-travelled code regions. This loop continues within a specified time budget, constantly refining the input set to explore untested program paths.

Algorithm 1 Coverage-Guided Fuzzing

```
1:  procedure FUZZ(targetProgram p, seedInputs I)
2:      Inputs ← I
3:      CoverageTracker ← getCoverage(p, Inputs)
4:      while time budget allows do
5:          inputTestCases ← selectFrom(Inputs)
6:          mutatedTestCases ← mutate(inputTestCases)
7:          testCoverage, testError ← execute(p, mutatedTestCases)
8:          if testError then
9:              reportError(mutatedTestCases, testError)
10:             optionally, terminate
11:         else if testCoverage ⊈ CoverageTracker then
12:             Inputs ← Inputs ∪ {mutatedTestCases}
13:         end if
14:         CoverageTracker ← CoverageTracker ∪ testCoverage
15:     end while
16: end procedure
```

2.1.2 Regression Fuzzing

In reaction to the Heartbleed vulnerability, OSS-Fuzz was initiated in 2016. To prove regressions have become a major class of bugs, the Regression Greybox Fuzzing project did an empirical study on OSS-Fuzz bug reports, as indicated by OSS-Fuzz project, 22,582 bugs were identified in 376 open-source projects, with 77.2% being regressions. Those regression bugs take an average of 68 days to be detected after the initial bug-introducing commit. The likelihood of a bug being a regression bug

increases with the number of reported bugs per project, indicating a growing challenge in managing software quality as projects evolve [25].

AFLChurn, or Regression Greybox Fuzzing (RGF) approach, is a state-of-the-art fuzzing methodology developed to efficiently identify bugs in software, particularly focusing on more recently or more frequently changed code. Regression Greybox Fuzzing acknowledges that most codes in a project are usually stable and seldom modified, thus there is no point in treating every code basic block equally. To improve the efficiency of fuzzing and seeking those regression bugs, Regression Greybox Fuzzing uses a Simulated Annealing-based Power Schedule inspired by the metallurgical process of annealing. It adjusts the temperature to balance between the exploration of new paths and the exploitation of known paths. Ant Colony Optimization (ACO)-based Byte-Level Power Schedule which mimics the behaviour of ants seeking paths to food, on the other hand, is used to choose which bytes to mutate in input data. Both methods help in assigning more energy to code segments that are more likely to uncover regressions when fuzzed [25].

2.2 Use-after-free

2.2.1 Introduction

Use-after-free (UAF) represents a specific class of software vulnerability, particularly common in programming languages that do not automatically enforce memory safety, such as C, C++ and Assembly. In these not-memory-safe languages, the responsibility for managing memory allocation and deallocation falls on the programmers, which involves handling memory on both the stack and the heap. Use-after-free occurs when a program does not properly clear or reset a pointer after freeing the memory it points to, thus an already freed memory will still be accessed by those pointers. This will potentially lead to various unpredictable behaviours, including program crashes and data corruption. In the context of fuzzing, addressing use-after-free issues is also crucial, as they can prevent a fuzzer from reaching deeper code paths. If a fuzzer encounters a use-after-free, it may cause the program to crash prematurely, reducing the fuzzer's ability to explore further and identify other potentially more severe vulnerabilities.

The allocation on the stack is static, which means the lifetime and the memory size are fixed at compile time and will be automatically controlled by the system. In contrast, the memory on the heap is allocated dynamically, mostly controlled by the programmers themselves. Even though this brings more flexibility, it also introduces the possibility of errors like use-after-free. This is where functions and operators such as `malloc`, `new`, `free`, and `delete` come into play. They are used to manually control memory allocation and deallocation on the heap. A use-after-free error occurs when a program continues to use a pointer after the memory it points to has been freed.

Apart from Use-after-free, Double-free is another type of memory management vulnerability.

Both of them are part of a broader category of software flaws known as memory safety errors. Double-free occurs when a program attempts to free an already freed memory space twice. Because the memory is already recycled and might have been reallocated and used elsewhere in the program, similar to use-after-free, double-free can cause program instability and crashes, or be exploited to perform arbitrary code or corrupt the memory of a process.

Both Use-after-free and Double-free have a same **precondition**, which means that they can only happens when memory deallocation (free) is triggered before the crash location is reached. Because the free function creates the state where memory is both marked as available for reallocation and still can be erroneously referenced by the program.

2.2.2 Real-world Examples

Use-After-Free vulnerabilities are not confined to obscure or poorly maintained software; they are prevalent even in renowned projects. For instance, an examination of the 'torvalds/linux' repository reveals several UAF issues, as evidenced by their respective Common Vulnerabilities and Exposures (CVE) entries. The CVE-2023-1652 affects 'torvalds/linux' package with versions between 5.14-rc1 and 6.2-rc5, with low attack complexity and privileges required, and received a 7.1 high CVSS score [15]. The use-after-free flaw was found in `nfsd4_ssc_setup_dul()` in `fs/nfsd/nfs4proc.c` in the NFS filesystem in the Linux Kernel. This issue could allow a local attacker to crash the system or it may lead to a kernel information leak problem.

The use-after-free bug can be explained using the CVE-2023-1652 we mentioned above as an example. In the original implementation of the `nfsd4_ssc_setup_dul` function, a use-after-free condition arises under certain circumstances. The problematic code is as follows:

Listing 2.1: UAF Bug-Inducing Code

```
/* ... previous code ... */
struct nfsd4_ssc_umount_item *work = NULL;
work = kzalloc(sizeof(*work), GFP_KERNEL);

try_again:
/* ... */
if (signal_pending(current) ||
    (schedule_timeout(20*HZ) == 0)) {
    kfree(work);
    return nfserr_eagain;
}
/* ... */
finish_wait(&nn->nfsd_ssc_waitq, &wait);
goto try_again;
/* ... following code ... */
```

In this snippet, the variable `work` is dynamically allocated memory, which may be freed within the conditional block if a signal is pending or a timeout occurs. After freeing `work`, the function may return an error code, indicating that the operation should be retried. However, if the `goto try_again` statement is executed before the function returns, the loop will restart, there is a time window where the `work` pointer is pointing to freed memory, and the code will attempt to use the now-freed `work` pointer, and any operations on it would constitute a UAF vulnerability, leading to undefined behavior.

Listing 2.2: UAF Vulnerability Fix in nfsd4_ssc_setup_dul Function

```
/* ... previous code ... */

if (signal_pending(current) ||
    (schedule_timeout(20*HZ) == 0)) {
    kfree(work);
    return nfserr_eagain;
}

/* ... following code ... */
```

The issue was due to the premature freeing of the `work` pointer, which could then potentially be used again in a subsequent iteration or by another part of the system. The patch addresses this by adding a call to `finish_wait()` before freeing the work variable, which would ensure that all operations that could be waiting on the wait structure are completed, thus any pending operations are completed before freeing the memory.

A use-after-free vulnerability in the real world can have a far-reaching impact, especially in those widely used software components. Another classic example is the CVE-2014-0160 Heartbleed Bug in OpenSSL, the widely used open-source cryptographic library by many open-source web servers, such as Apache and Nginx. The Heartbleed Bug was essentially a Use-after-free vulnerability in the TLS/DTLS heartbeat extension. This vulnerability allows malicious users to trick a vulnerable web server into sending sensitive information, such as important usernames and passwords, the victim will bleed out those data through its heartbeat requests, and such data breaches will result in huge losses.

Chapter 3

Design

3.1 Precondition-guided Regression Fuzz Testing

3.1.1 Workflow Explaination

As illustrated in Figure 3.1, in a typical workflow of *coverage-guided fuzz testing*, a fuzzer feeds numerous inputs (test cases) to the target program and intends to find the specific inputs that have the ability to crash the program. The generation of test cases is an iterative process that evolves from mutating known interesting inputs — referred to as seeds — from earlier fuzzing iterations. The fuzzer starts with an initial collection of seed inputs and applies various mutations to these seeds to produce new test cases.

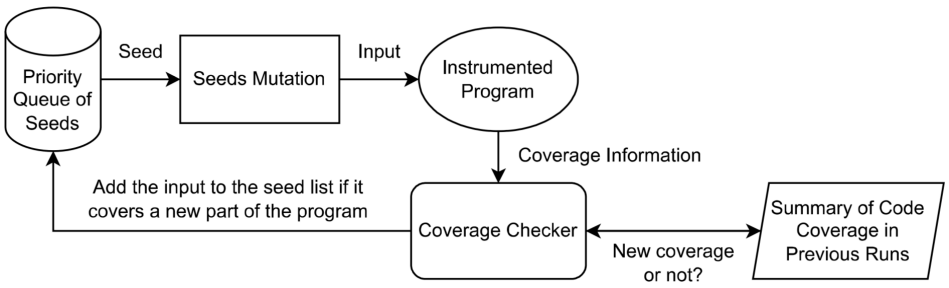

Figure 3.1: Coverage-guided Fuzzing Workflow

An input is considered "interesting" if it triggers the execution of previously unexplored paths within the program, such as detecting new edges between basic blocks, those interesting inputs will be added to the seeds queue. The code coverage information is collected based on the instrumentation, when a new edge is executed, the corresponding code coverage hit count will increase.

AFL-like fuzzers keep a bitmap data structure to keep the code coverage summary in the previous runs, and compare the current path log against past records to check whether the input hits new coverage [10].

In order to better identify regression vulnerabilities, regression greybox fuzzing implemented a variant of AFL called AFLChurn. During the code instrumentation phase, AFLChurn instruments the target program to monitor each basic block's `age` and `churn` information indicating how recently and how frequently it has been changed [25]. During the seed mutation phase, when the initial seeds are mutated to produce new inputs to test the program, AFLChurn deploys an ACO-based byte-level power schedule strategy to prioritize mutations on the most effective bytes.

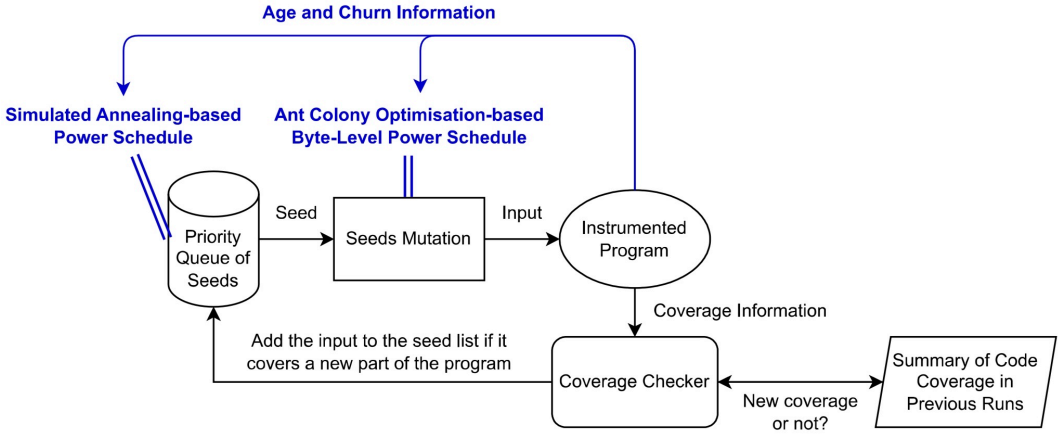

Figure 3.2: Regression Fuzzing Workflow

Evolving from regression greybox fuzzing, our project designs a precondition-guided regression fuzzing technology. Its workflow enhances the vanilla regression fuzzing process by integrating an additional analysis of the precondition map, as shown in Figure 3.3. As illustrated in Section 2.2.1, use-after-free cannot happen without triggering preconditions (e.g., free). Thus the workflow augments regression greybox fuzzing to incorporate the precondition information generated from the instrumented program. It begins by instrumenting the target program's basic blocks and focuses on blocks where memory deallocation occurs and blocks that have been recently modified. The instrumentation enables the fuzzer to track when the program enters these blocks, creating a runtime log of precondition events called **precondition map**. The fuzzer uses the precondition map to determine if an input can reach the precondition to trigger use-after-free bugs, this not only includes reaching new parts of the program but also pays special attention to triggering new preconditions. Inputs that satisfy new preconditions are also considered "interesting" and will be added to the priority queue for further mutations and deeper exploration.

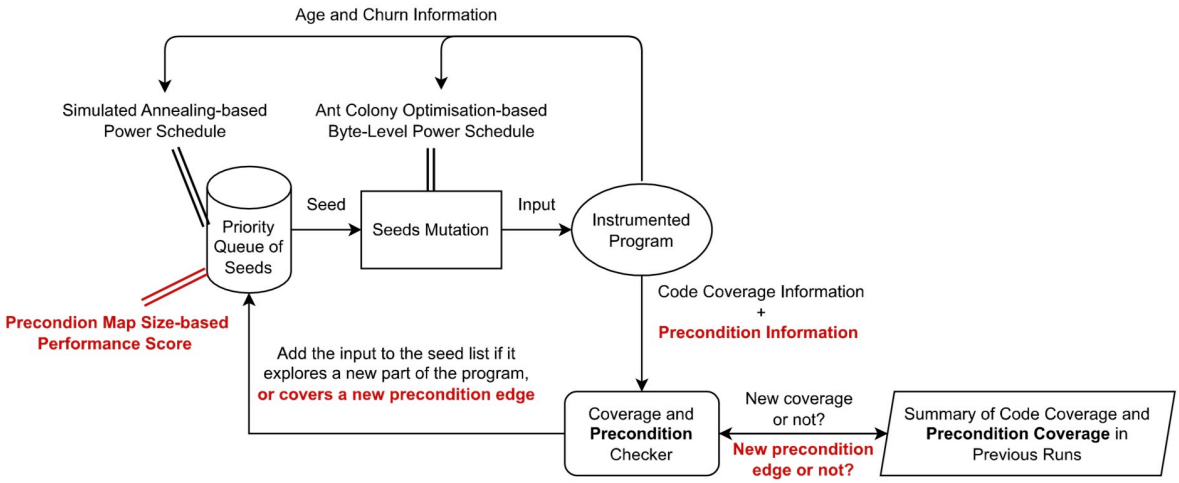

Figure 3.3: Precondition-guided Regression Fuzzing Workflow

This enriched dataset enables the fuzzer to make more informed decisions about which inputs to retain for further mutation and analysis, particularly favoring those that trigger new preconditions or explore uncharted execution paths.

3.1.2 Scoring Mechanism

Apart from adding interesting test cases to the queue of seeds, AFL-like fuzzers also employ a distinctive scoring mechanism to prioritize inputs (test cases) during the fuzzing process, the scoring function is critical because it determines the "interest" level of a given input by assigning it a performance score, the fuzzer will choose input with a higher score more frequently in the subsequent fuzzing cycles.

In our project, we augment AFL's `calculate_score` function which is designed to evaluate the potential of each queue entry to uncover new bugs. It assigns a performance score (`perf_score`) that influences how frequently the entry will be selected for future fuzzing iterations. The code snippet in Appendix A showed the heuristic we used in our scoring system. We define `precondition map size` as how many different precondition edges were hit by the input, a larger-than-average precondition map size suggests the input is potentially more likely to trigger a use-after-free bug, thus those inputs will gain higher scores. The performance score adjustment is based on the principle that inputs leading to a larger precondition map size are considered more valuable:

- Inputs that result in a precondition map size significantly larger than the average receive a score multiplier, enhancing their selection likelihood.
- Conversely, inputs associated with a smaller precondition map size are considered less interesting and are assigned a lower multiplier, reducing their fuzzing priority.

This augmented version of the scoring system will be combined with simulated annealing-based power schedule. The fuzzer uses the scores calculated for each seed to decide how to distribute the fuzzing efforts. The inputs that cover more UAF preconditions are allocated more fuzzing time.

3.2 Precondition Map Construction

In the context of software vulnerabilities, a precondition refers to the set of conditions that must be met for a bug or security flaw, like a use-after-free (UAF) vulnerability, to manifest or be exploitable. For a UAF vulnerability, the preconditions typically include:

- **Memory Allocation:** There must have been dynamic memory allocation that was once valid and accessible via a pointer.
- **Memory Deallocation:** The allocated memory must be freed, which means it is no longer valid for use.
- **Dangling Pointer:** After the memory is freed, there must still exist a pointer (now a dangling pointer) that references the already deallocated memory.
- **Improper Handling:** The program logic must erroneously allow for the reuse or access of the freed memory via the dangling pointer without proper checks or reallocation.
- **Triggering Condition:** There must be a specific sequence of operations or inputs that leads to the use of the dangling pointer, triggering the UAF condition.

In our case, we are specifically trying to identify regression use-after-free and double-free vulnerabilities, which present significant challenges in software maintenance and security. Our approach to detecting those vulnerabilities relies on the construction of the precondition map which leverages two cyclic queues: one for basic blocks that contain deallocation such as `free` functions and another for basic blocks that contain newly changed codes in recent commits. This method is based on the understanding that recent changes to the codebase might improperly handle previously freed memory areas, and will introduce regression use-after-free bugs.

AFL uses a concept called "edge coverage" to construct its bitmap, which is the central component of its fuzzing strategy. In our detection strategy, we use the concept of precondition map, which is similar to the bitmap in AFL, while it tracks the edges between `free` blocks and recent commit blocks. We implement two strategies to construct the precondition map, which results in two versions of instrumentation. It operates under two potential scenarios that could introduce use-after-free (and double-free) bugs through new commits:

1. **Precondition Version 1:** As shown in Figure 3.4, given the nature of UAF bugs, there is a substantial likelihood that a previously freed memory area might be accessed erroneously.

In such cases, our methodology involves computing the hash of the basic block identifier along with the identifiers of all the free blocks within the free list queue. This computation occurs each time a recently changed basic block is executed. The resultant hash is then used to increment the corresponding count in the precondition map, thereby signalling a potential UAF scenario.

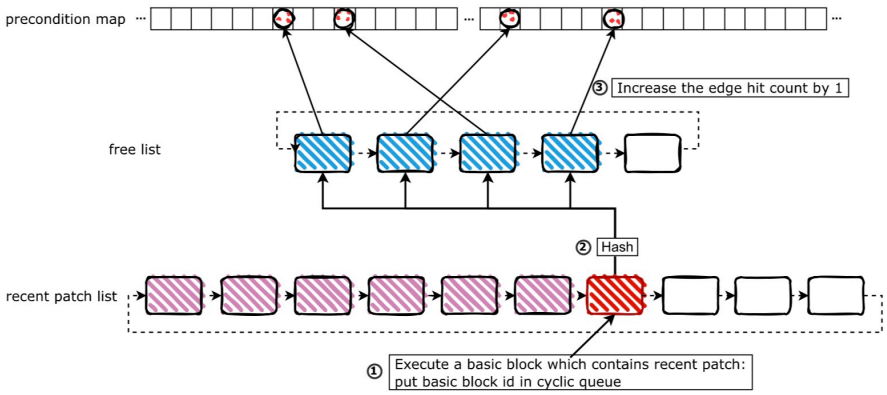

Figure 3.4: Precondition Version 1

2. **Precondition Version 2:** As shown in Figure 3.5, this method implements a bidirectional link strategy. When executing a basic block containing a `free()` function, we compute the hash using the basic block's identifier and all the basic blocks in the recent-commits list. Conversely, execution of a basic block that has been recently modified prompts the same action as in Version 1, linking it to the free blocks. This dual approach aids in capturing both UAF and double-free bugs by observing interactions between free operations and recent code changes.

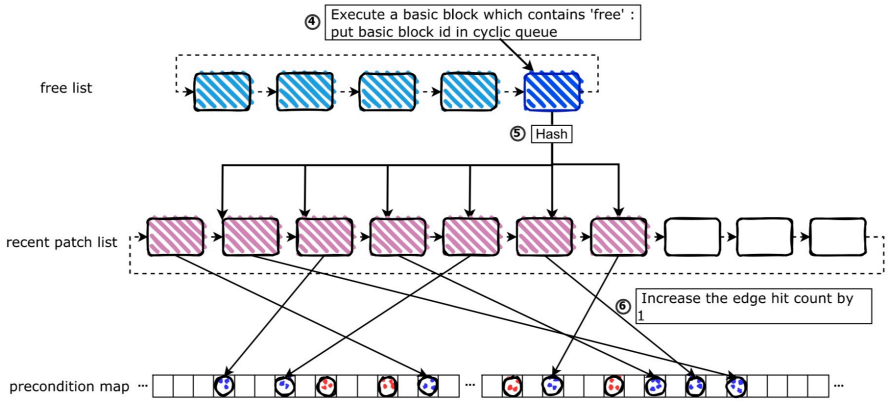

Figure 3.5: Precondition Version 2

The cyclic queues track and update the status of memory operations and code modifications, providing a temporal context to the detection mechanism. When a potential UAF is detected, the precondition map offers a historical insight by correlating current execution paths with past

memory deallocations and modifications. This correlation is key to identifying regression bugs, as it highlights how new changes might reintroduce previously patched or unnoticed vulnerabilities.

Our detection strategy is designed to be both robust and efficient. By focusing on recently changed code and free operations, we narrow down the detection scope to the most probable areas where regression bugs might occur. This targeted approach enables us to utilize computational resources efficiently and rapidly identify vulnerabilities that could compromise system stability or security.

3.3 Commits-based Instrumentation Period Selection

This project operates under the assumption that the use-after-free bugs we are examining are regression bugs. To effectively address these, we give precedence to inputs that activate a greater number of precondition edges. However, a challenge arises when considering the entire git history for instrumentation. Such an exhaustive approach might result in every basic block being linked to a segment of commit history, thereby diluting the effectiveness of the precondition map in the fuzzing process. To optimize fuzzing, our strategy aims to cover the period encompassing the bug-introducing commit. Instrumenting too many commits preceding the bug root can overload the fuzzing process and diminish the utility of the precondition map.

In real-world scenarios, it's often the case that companies and developers are unaware of the existence or the specific introduction point of a bug within their projects. Moreover, the application of a uniform instrumentation standard across various projects is not feasible due to the differences in project scales and their respective stages of development. Recognizing these practical constraints, we have opted for a commit-amount-based instrumentation approach. This method tailors the instrumentation of basic blocks in relation to the number of current commits in a project, allowing for a more customized and effective analysis.

Our instrumentation methodology is designed to maximize the likelihood of covering the bug-introducing commit while avoiding the inclusion of excessive, irrelevant commits:

- For projects with a commit history ranging from 0 to 3,333, we set the instrumentation proportion at 15%. This percentage is carefully chosen to provide sufficient coverage without overwhelming the fuzzing process.
- For projects with more than 3,333 commits, we cap the instrumentation at 500 commits. This cap is designed to maintain manageability and efficiency in larger, more complex project histories.

This approach balances the need for comprehensive bug analysis with the practical limitations of fuzzing large commit histories.

Chapter 4

Implementation

4.1 Source Code Compiler-level Instrumentation

Instrumentation, a prevalent technology in both static and dynamic analysis, is utilized to analyse and potentially modify the instructions within the source code. The process requires scanning each instruction in the module to determine if it corresponds with any predefined instrumentation rules. When a match is identified, new code will be inserted as specified by the rule. This can include additional instructions or modifications to existing ones, tailored to monitor or change the behavior of the program for analysis purposes. The output of this instrumentation process is a modified program, which retains the original functionality while incorporating the changes for new functionalities. In our case, we are going to do compiler-level instrumentation based on `LLVM mode` of AFL. The instrumentation is at the LLVM Intermediate Representation (IR) level and inserts instrumentation codes at the beginning of each basic block [11].

LLVM [19] is an open-source project offering compiler technologies that are independent of specific programming languages and target architectures. The project encompasses multiple components: the LLVM Intermediate Representation (commonly abbreviated as LLVM IR or simply LLVM) for interpreting source code, the LLVM Core libraries which are used for code generation, optimization, and transformation, and various tools including the Clang compiler and the LLVM optimizer (opt). A critical element of the LLVM system is the LLVM Pass Framework, which is important for the transformation and optimization of LLVM code [22]. In the context of fuzzing instrumentation, LLVM IR is the fundamental component serving as a middle ground between source code and machine code, thus playing an important role in the context of the LLVM Pass framework. These passes can analyze or modify the LLVM IR, enabling a range of optimizations and analyses to be conducted at different stages of the compilation process.

The Clang compiler, while not primarily designed for general instrumentation, employs in-

strumentation through LLVM. It includes a range of sanitizers that implement runtime checks to detect various errors and undefined behaviors. Notably, it features AddressSanitizer [17], a tool for detecting memory errors including use-after-free and double-free we focusing on in our project. AddressSanitizer works by instrumenting each memory access with a verification step against a shadow memory. This shadow memory holds metadata about the corresponding program memory, especially the 'poisoned redzones', which are areas of memory that are flagged as off-limits for access [22].

To understand how instrumentation benefits fuzzing procedure and illustrate the use of LLVM IR in instrumentation, we take a look at American Fuzzy Lop (AFL)'s bit map construction. AFL employs a classic and effective instrumentation strategy, and leverages LLVM IR to capture detailed branch coverage and coarse hit counts for branches taken. At compile-time, each branch point in the code is augmented with the following snippet:

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

Here, `cur_location` is assigned a random value at compile-time. This randomness aids in maintaining a uniform distribution of the XOR operation output and simplifies the instrumentation of complex projects. The `shared_mem[]` is an array backed by a 64-kilobyte shared memory region, which the instrumented binary accesses. Each byte in this map signifies a hit count for a specific tuple of source and destination branches. The map size is carefully chosen to minimize collisions while accommodating the number of branches typical in most targets, which ranges from 2,000 to 10,000 discoverable branch points, and is also small enough to allow rapid analysis and fit comfortably within the L2 cache [24].

The AFL's instrumentation approach keeps more information about the program's execution path than mere block coverage. It can differentiate between sequences of branches, thereby uncovering subtle bugs related to state transitions, which are often very important in identifying security vulnerabilities.

4.1.1 Instrumenting Projects with Different Scale

In our work, we have established a benchmark involving seven open-source C language projects, with a specific focus on identifying and analyzing the roots of bugs within these projects. This analysis aims to discover patterns in commits that introduce bugs, providing valuable insights for future bug prevention strategies.

Among the projects analyzed, *gifsicle* stands out as it lacks a clear bug-introducing commit. Despite an exhaustive review of its development history stretching back to 1999, the use-after-free

bug remains there. In contrast, *nasm* exhibited a bug that introduced 454 commits before the total of 3,566 commits logged at the time of reporting. Similarly, for *readelf*, the use-after-free bug was found to have occurred 167 commits prior out of a massive 96,395 total commits. *Jpegoptim* had a shorter timeline, with the bug root appearing just 7 commits before out of 154 total. For the projects *mjs-b1b6eac* and *mjs-5c9fd64*, there were 85 and 213 commits, respectively, following the bug introduction, out of 573 and 454 total commits at the point of bug reporting. After analyzing the bug-introducing pattern in the benchmark projects, we confirm that the period selection method we designed and explained in section 3.3 can cover bug-introducing commit in most cases.

To instrument the source code while focusing on the interesting commits, our project utilizes Git functions for identifying changes in the basic blocks, the pseudocode is shown in Appendix B. We first execute a Git command `git rev-list -count HEAD` to determine the total number of commits affecting the file. Based on this count, the code decides how many commits to instrument. The script sequentially calls `git log -> git show -> git diff` to retrieve the commit hashes within this defined range, determines the specific lines that have been changed in each commit, and gets the related lines in the HEAD commit. Such line-level changes would be linked to basic blocks to identify which basic block is affected by recent commits.

4.1.2 Free-based Instrumentation

As discussed in Chapter 3.2, the generation of a precondition map is a critical process that requires the strategic instrumentation of certain basic blocks within a program—specifically those that execute memory deallocation routines like `free()`. The LLVM IR offers a rich set of APIs that allow for the precise manipulation of instructions and control flows. Our process of instrumentation is implemented based on the AFL's LLVM mode pass and includes the insertion of additional code at the location of each `free()` call. These additional codes are doing the job of forming the cyclic queue and precondition map.

In practice, memory deallocation functions are not always explicitly named '`free()`'. We assume that any function responsible for memory deallocation includes 'free' within its name. For instance, in CVE-2023-1652 (refer to Section 2.2.1), the function is named `kfree()`, which adheres to this naming convention. This heuristic has been effectively utilized in our project for the identification of memory deallocation calls. Our implementation traverses each basic block within a function and inspects each instruction for a call to a memory deallocation function. Upon identifying a call that contains 'free' in the function name, it logs this occurrence and sets a flag indicating the presence of a `free()` call within the basic block.

4.2 Cyclic Queue and Precondition Map Size Selection

As explained in Section 3.2, our approach utilizes two distinct cyclic queues: the free list and the recent patch list. These queues are fundamental in monitoring the dynamic execution state of the program, focusing specifically on memory deallocation and newly introduced code changes, respectively. The size of these queues is not arbitrary; they are carefully chosen to balance the efficiency and effectiveness of the fuzzing process. The recent patch list is configured to contain *ten* entries. This size is chosen to prevent the distraction of attention across too many changes while still capturing a meaningful window of recent code activity. Conversely, the free list is constrained to *five* entries. By focusing on the last five deallocations, the fuzzer prioritizes the most recent and relevant memory operations that could lead to use-after-free vulnerabilities. The size is smaller than the recent patch list because the set of basic blocks in the new commits is significantly larger than functions which intend to free memory.

The choice of queue sizes—10 for the recent patch list and 5 for the free list—implies a strategic prioritization, directing more resources towards monitoring code changes. The cyclical nature of these queues ensures that the fuzzer's attention is continuously updated with the most recent and pertinent code and memory operation information, aligning with the fast-paced evolution of the code during the fuzzing campaign. This empirically informed sizing strategy enhances the fuzzer's capability to discover and prioritize the exploration of inputs that are likely to uncover use-after-free vulnerabilities.

The size of the precondition map is selected considering the tradeoff of accuracy and speed, which is a 64KB shared memory segment [23], in the context of fuzzing, it allows the fuzzer and the target program to communicate, the target program will update the precondition map in real-time during execution and the fuzzer will be able to read it. Essentially, the precondition map is a variant of bitmap, where each byte corresponds to a particular transition between basic blocks in the program execution. The chosen map size of 64Kt works well for a wide range of target program sizes, it has been proven by AFL to be small enough to be quickly analyzed and fit within the CPU's L2 cache, which assures it with an efficient performance. At the same time, it has a relatively low rate of hash collision, where different precondition edges are mapped to the same offset in the map. While the collision is inevitable, it won't significantly impact the overall fuzzing effectiveness.

Chapter 5

Evaluation

Our *main hypothesis* is that a fuzzer, when strategically directed towards areas of code that transit between recent changes and frequent memory deallocation operations, will be more effective in uncovering regression use-after-free vulnerabilities. In our experiment, we test our precondition-guided fuzzing approach based on this hypothesis.

5.1 Research Questions

RQ1 Does directing our fuzzing campaign towards more precondition edges improve the efficiency of greybox fuzzing to identify regression use-after-free bugs? We conduct a comparative analysis between our fuzzer against AFL and AFLChurn to measure the *Time-to-Exposure*(TTE).

RQ2 Which heuristic is more effective, i.e., version 1, which constructs a precondition map with single-directional transitions, or version 2, which accounts for bi-directional transitions? We evaluate the performance of our fuzzer under two distinct guiding principles: (a) unidirectional precondition map, and (b) bidirectional precondition map.

5.2 Benchmark Subjects

To construct a suitable and valuable benchmark, we contribute in the following ways:

- **New benchmarks.** In our case, the benchmark required Git commit information, because our project is based on AFLChurn which uses `git blame` to identify the commit for a given line [25], and the benchmark project should have been reported use-after-free vulnerability.

There doesn't exist any mature benchmark to identify regression use-after-free and meet all our requirements. We thus construct our own regression benchmarks as per the selection criteria stated.

- **Regression Analysis.** We extract the relevant information from all seven projects, including when is the use-after-free bug first reported, manually identify the root cause commit that introduced the use-after-free bug, and create scripts to automatically calculate the commit distance from the reported commit and the bug-introducing commit.

Table 5.1 presents the details of the selected subjects for our benchmark. We have chosen use-after-free bugs from seven distinct open-source C projects hosted on GitHub, specifically selected to represent a variety of scales. *Test Commit* column refers to the particular commit we used for fuzz testing, which is typically the commit where the use-after-free bug was initially reported. *Regression* column details the commit at which the bug was introduced; this was pinpointed manually by regressive analysis using the use-after-free proof-of-concept. *Distance* is calculated as how many commits between the bug reported commit and bug introducing commit. Among these projects, *giscile* stands out as the only one for which the bug-introducing commit could not be identified, despite tracing back to 1999. This suggests that the bug may have been present since the inception of the project.

Table 5.1: Regression Use-after-free Bugs in Five Open-source C Programs

Project	Test Commit	Regression	Reported	Distance	Commits Count	Bug Type
readelf	923c6a7	0a59dec	31.Dec'18	167	96,395	UAF
mjs	b1b6eac	e0faf9f	31.Dec'21	85	573	UAF
jpegoptim	d23abf2	83dfad1	24.May'18	7	154	DF
nasm	9516cf3f	f640b61	29.Jun'17	454	3,566	UAF
mjs	9eae0e6	5c9fd64	26.Dec'17	213	454	UAF
xmllint	798bdf13	0e1a49c	21.Mar'22	245	5,086	UAF
gifsicle	fad477c	not found	04.Jan'18	INF	814	DF

5.3 Setup and Infrastructure

The experiments are designed for full reproducibility, with the source code publicly available on GitHub. To mitigate the influence of random factors, each experiment was iterated multiple times, with repetitions ranging from 5 to 20, based on the Time-to-Exposure (TTE) metrics observed. The default configuration for fuzzing allocated one CPU core per campaign, and the experimental was deployed on a machine equipped with a 12-core CPU, 32GB of RAM, and 300GB of available storage space. The operating system used was Ubuntu 22.04 64-bit.

We didn't use platforms such as FuzzBench for our evaluation, instead, it was executed on a

consumer-grade machine. It is crucial to acknowledge that the outcomes of the experiment could be influenced by the limitations of the hardware resources available.

5.4 Experiment Results

Scoring System Effectiveness When new paths have been discovered, the fuzzer will calibrate it to collect useful fuzzing information for the fuzzing campaign. As part of our evaluation, we analyzed the proportion of entries that activated precondition edges. In our benchmarks, while testing *readelf*, *nasm*, *jpegoptim*, and *mjs*, all entries exhibited precondition edges. However, in the case of *gifsicle*, none of the seed entries activated any precondition edges. Similarly, for the largest project *readelf*, the majority of entries did not engage any precondition edges.

Using *readelf* as an illustrative case, we demonstrate the effectiveness of our precondition map and scoring mechanism in enhancing regression use-after-free fuzzing. The most optimal results were observed with unidirectional precondition fuzzing, achieving an average TTE of 409 seconds. In the initial seed set, none of the 15 test cases triggered a precondition transition. Consequently, in a typical fuzzing campaign targeting *readelf*, the first few minutes (approximately the first four minutes) do not see any seed entries generated with bytes in precondition map being set. However, from the fourth minute onward, once inputs activating precondition edges are identified, the first crash tends to occur within 40 seconds. This rapid detection aligns with our expectations and shows the efficiency of the scoring mechanism introduced in Chapter 3.1.

The observed phenomenon where some test cases do not engage any precondition transitions in the *readelf* benchmark can be attributed to its belonging to the extensive *binutils* project. At the time the bug was reported, *binutils* had a total of 96,395 commits. Within such a large project scope, the 500 commits we consider for instrumentation may not be directly associated with the *readelf* executable. Consequently, during execution, the likelihood of executing certain lines from recent patches is considerably lower compared to a mid-sized project. Although memory deallocation operations are an inevitable aspect of program execution, if the test cases randomly avoid intersecting with these recently patched segments, they may not trigger the precondition edges.

Evaluation Result Time-to-Exposure (TTE) quantifies the duration of the fuzzing campaign until the first crash is exposed by inputting certain test cases [3]. Table 5.2 provides a detailed summary of our experimental results. In this discussion, we refer to our fuzzer variants as *UAF-Bidirection* and *UAF-Unidirection*, which correspond to the two distinct methodologies for precondition map construction. In our tests, all fuzzing campaigns conducted by these fuzzers were successful. For comparative analysis, we rely on the average TTE, denoted as μTTE. As evidenced in Table 5.2, our fuzzing approaches demonstrate superior performance in most scenarios when compared to AFL and AFLChurn.

Table 5.2: μTTEs of Our Fuzzer against AFL and AFLChurn

Bug	AFL		AFLChurn		UAF-Bidirection		UAF-Unidirection	
	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)
readelf	10	481	10	507	10	419	10	409
mjs-b1b6eac	5	5,291	5	4,226	5	551	5	2,775
jpegoptim	20	33	20	30	20	23	20	23
nasm	10	4,665	10	1,879	10	4,504	10	1,561
mjs-9eae0e6	15	110	15	103	15	70	15	107
xmllint	5	2,754	5	2,614	5	2,776	5	2,767
gifsicle	10	237	10	363	10	371	10	333
Best Performance	1		1		3		3	

Across multiple runs, both UAF variants consistently achieved lower average Time-to-Exposure (μTTE) values, signaling a quicker discovery of vulnerabilities. For instance, in the *readelf* project, UAF-Unidirection reported a μTTE of 1,561 seconds, significantly outperforming AFL's 4,665 seconds and AFLChurn's 1,879 seconds. This trend persisted across various benchmarks, UAF-Unidirection shows the best performance in projects including *readelf*, *jpegoptim*, and *nasm*, and UAF-Bidirection shows the best performance in projects including *mjs-b1b6eac*, *jpegoptim*, and *mjs-9eae0e6*. UAF-Bidirection and UAF-Unidirection demonstrated superior performance as indicated by the `Best Performance` row, where they collectively scored the majority of leading positions. After calculating the geometric mean of the speedup ratios, UAF-Bidirectional demonstrated a 49% enhancement in bug detection time over AFL, and a 31% improvement compared to AFLChurn. UAF-Unidirectional reported a 32% improvement over AFL and a 17% improvement over AFLChurn. These results indicate that UAF-Bidirectional has a more pronounced speedup capability.

Notably, in the *gifsicle* case, where the bug does not fall under the category of a regression bug, AFL delivers the best performance without surprises. Conversely, for *xmllint*, AFLChurn emerged as the top performer with a very slight advantage. This closely contested outcome can be attributed to the quality of the seeds we choose. Even though AFLChurn is only the best performer of one project, it outperforms AFL in most cases, which demonstrates its effectiveness in regression grey-box fuzzing. The overall performance across different projects indicates that our precondition-guided approach significantly enhances the efficiency of identifying regression use-after-free bugs, with UAF-Bidirectional being particularly noteworthy.

Chapter 6

Related Work

UAF Directed Greybox Fuzzing. *Directed Greybox Fuzzing* (DGF) [3] performs stress testing by using a power schedule to assign more energy to seeds that are closer to potentially vulnerable target program locations. Nguyen et al. developed UAFuzz, an efficient binary-level directed fuzzer tailored for UAF [13]. Built on top of AFL-QEMU, UAFuzz is one of the very few fuzzers with the ability to fuzz binary code. It recognizes bug traces of a UAF crash as a target, computes the target distance for each basic block and selects seed based on target similarity. Even though UAFuzz showed effectiveness in UAF detection and its ability to work on the binary level is impressive, it overlooks the fact that regressions are the main cause of bugs nowadays.

Prevalence of Regression Bugs. Our work is based on the assumption that recent patches tend to have a higher potential to introduce bugs. The prevalence of regression bugs is well-known in the vulnerability prediction community. In an empirical study by Zhu et al. involving analysis of 22,582 bug reports from the OSSFuzz platform, it was found that a substantial 77.2% of the bugs were regressions. Furthermore, the likelihood of encountering regression bugs increases as the project matures; the study suggests that the 1000th bug being reported in a project has a probability exceeding 99% of being a regression [25].

Regression Greybox Fuzzing. In order to fuzz active software projects and search for regression bugs, regression greybox fuzzing developed AFLChurn to direct fuzzer towards code that is changed more recently or more frequently [25]. AFLChurn makes use of the code versioning system instead of the analyzed call graph and control flow graphs, allowing it to focus on a large range of commits. The byte-level power schedule based on Ant Colony Optimization analyses each byte's influence on the seed's fitness and assigns more energy to the most interesting bytes to be mutated. However, the regression greybox fuzzing technology is quite generic and does not consider the specificities of precondition, therefore is blind to UAF bugs, which widely exist in non-memory safety programs

and are among the most critical exploitable vulnerabilities.

UAF Fuzzing Benchmark. The programs contained in Juliet Test Suite are too small, which lack the diversity of program scale; According to the experimental study, bugs in more active and comprehensive projects have a higher probability to be regression [4]. Among the fifteen bugs in the regression benchmark constructed by Regression Greybox Fuzzing project, OSSFuzz reports reveal that only two of them are categorised as Use-After-Free bugs [25]. Moreover, popular fuzzing benchmarks [5, 7, 16] in the market all have similar limitations, in that they don't contain enough UAF database. UAF Directed Greybox Fuzzing constructed its use-after-free benchmark [20] which is built on real-world bugs. However, the nature of UAFuzz determines that it ignores versioning information of projects, which is crucial in a real-world development environment and is necessary for evaluation of our project. Regarding all the limitations we mentioned in previously constructed benchmarks, we constructed our own benchmark, which contains seven open-source C projects with their Git history available.

Chapter 7

Conclusion

In this report, we introduced precondition-guided fuzzing to enhance the effectiveness of identifying regression Use-After-Free (UAF) and Double-Free (DF) vulnerabilities. We first discussed the preconditions necessary for triggering these regression UAF and DF vulnerabilities: recent commits inadvertently used a memory pointer that had already been freed, or altered the control graph flow in such a way that incorrect access to freed memory became possible.

We instrument source code and enable the tracing of transitions between newly changed codes and free operations with a structure called a precondition map, the input with the ability to enable precondition transition will be added into seeds, and the inputs cover more precondition edges will be given more CPU time to explore by the fuzzer. Developed based on regression grey-box fuzzing, our methodology also leverages the strengths of a simulated annealing-based power schedule and an Ant Colony Optimization-based byte-level power schedule strategy to prioritize inputs which execute more recently and frequently changed code areas.

Through rigorous evaluation involving a benchmark of seven use-after-free and double-free bugs in open-source C programs, our method demonstrated superior performance over AFL and AFLChurn. This can be attributed to its ability to focus on recent code changes and apply a Use-After-Free precondition-guided approach, thereby unearthing vulnerabilities that would likely be missed by conventional methods.

In summary, the regression precondition-guided fuzzing method represents a significant advancement in discovering regression Use-After-Free (UAF) and Double-Free (DF) vulnerabilities. Moreover, it possesses the ability to be expanded to a broader range of software vulnerabilities in the future.

Bibliography

[1] Dave Aitel. "An introduction to spike, the fuzzer creation kit". In: *presentation slides), Aug* 1 (2002).

[2] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. "Predicting bug-fixing time: A replication study using an open source software project". In: *Journal of Systems and Software* 136 (2018), pp. 173–186. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.02.021. URL: https://www.sciencedirect.com/science/article/pii/S0164121217300365.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 2329–2344.

[4] Tim Boland and Paul E Black. "Juliet 1. 1 C/C++ and java test suite". In: *Computer* 45.10 (2012), pp. 88–90.

[5] Brian Caswell. *Cyber Grand Challenge Corpus*. Ed. by Lunge Technology. URL: http://www.lungetech.com/cgc-corpus/.

[6] Zhen Yu Ding and Claire Le Goues. "An empirical study of oss-fuzz bugs". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 131–142.

[7] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. "Lava: Large-scale automated vulnerability addition". In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 110–121.

[8] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. "GREYONE: Data Flow Sensitive Fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/gan.

[9] Kevin J Hoffman, Patrick Eugster, and Suresh Jagannathan. "Semantics-aware trace analysis". In: *ACM Sigplan Notices* 44.6 (2009), pp. 453–464.

[10] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. "Instrim: Lightweight instrumentation for coverage-guided fuzzing". In: *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. 2018, p. 40.

[11] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. “InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 568–580. DOI: 10.1109/ASE51524.2021.9678671.

[12] Barton P Miller, Lars Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[13] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. “Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities”. In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 47–62. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/nguyen.

[14] Oulu University Secure Programming Group. *PROTOS Security Testing of Protocol Implementations*. http://www.ee.oulu.fi/research/ouspg/protos/. Accessed: Date-of-access. Year.

[15] Red Hat, Inc. *CVE-2023-1652*. https://access.redhat.com/security/cve/cve-2023-1652. Accessed: [insert date of access]. 2023.

[16] *Rode0day*. https://rode0day.mit.edu/. 2020.

[17] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “{AddressSanitizer}: A fast address sanity checker”. In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[18] Armstrong A Takang and Penny A Grubb. “Software maintenance: concepts and practice”. In: (1996).

[19] llvm-admin team. URL: https://llvm.org/.

[20] *UAF fuzzing benchmark*. https://github.com/strongcourage/. 2020.

[21] Hans Van Vliet, Hans Van Vliet, and JC Van Vliet. *Software engineering: principles and practice*. Vol. 13. John Wiley & Sons Hoboken, NJ, 2008.

[22] Martina Vitovská. “Instrumentation of LLVM IR”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2018.

[23] Hang Xu, Zhi Yang, Xingyuan Chen, Bing Han, and Xuehui Du. “BitAFL: Provide More Accurate Coverage Information for Coverage-guided Fuzzing”. In: *3rd International Conference on Management Science and Software Engineering (ICMSSE 2023)*. Atlantis Press. 2023, pp. 521–530.

[24] Michal Zalewski. URL: https://afl-1.readthedocs.io/_/downloads/en/latest/pdf/.

[25] Xiaogang Zhu and Marcel Böhme. “Regression Greybox Fuzzing”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2169–2182. ISBN: 9781450384544. DOI: 10.1145/3460120.3484596. URL: https://doi.org/10.1145/3460120.3484596.

Appendix A

Performance Score Augmented to AFL

The scoring mechanism is integral to the fuzzing strategy, as it determines the priority given to each test case. Below is the code snippet that illustrates our customized scoring function, adapted from AFL's methodology.

Listing A.1: Custom Scoring Function

```
static u32 calculate_score(struct queue_entry* q) {
  // ... (other parts of the scoring function)

  u32 avg_precondmap_size = total_precondmap_entries > 0 ?
    total_precondmap_size / total_precondmap_entries : 0;
  u32 perf_score = 100;

  // Adjust score based on precondition map size
  if (avg_precondmap_size > 0) {
    if (q->precondmap_size * 0.3 > avg_precondmap_size) perf_score *= 3;
    else if (q->precondmap_size * 0.5 > avg_precondmap_size) perf_score *= 2;
    else if (q->precondmap_size * 0.75 > avg_precondmap_size) perf_score *= 1.5;
    else if (q->precondmap_size * 3 < avg_precondmap_size) perf_score *= 0.25;
    else if (q->precondmap_size * 2 < avg_precondmap_size) perf_score *= 0.5;
    else if (q->precondmap_size * 1.5 < avg_precondmap_size) perf_score *= 0.75;
  }

  // ... (rest of the scoring function)

  return perf_score;
}
```

Appendix B

Identify Changed Basic Blocks

Algorithm 2 Calculate Line Changes Based on Commit Count in Git Repository

1: **Input:** *relativeFilePath*, *gitDirectory*, *changeSig*
2: **Output:** *fileToLineToChangeMap*
3: **procedure** CALCULATECOMMITBASEDCHANGES
4: *fileToLineToChangeMap* ← {}
5: *totalCommits* ← count total commits for *relativeFilePath*
6: **if** *totalCommits* ≤ 3,333 **then**
7: *commitsToAnalyze* ← 0.15 × *totalCommits*
8: **else**
9: *commitsToAnalyze* ← 500
10: **end if**
11: *commitList* ← retrieve last *commitsToAnalyze* commit hashes
12: **for** each *commit* in *commitList* **do**
13: *changedLinesCurCommit* ← get changed lines in *commit*
14: *linesToChanges* ← map to current HEAD
15: Normalize *linesToChanges* and update *fileToLineToChangeMap*
16: **end for**
17: Handle errors
18: **end procedure**