# Unearthing Hidden Bugs: Harnessing Fuzzing With Dynamic Patching in FlakJack

by

Gokulkrishna Praveen Menon

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2023 by the
Graduate Supervisory Committee:

Tiffany Bao, Co-Chair
Yan Shoshitaishvili, Co-Chair
Adam Doupe

ARIZONA STATE UNIVERSITY

August 2023

ABSTRACT

This thesis presents a study on the fuzzing of Linux binaries to find occluded bugs. Fuzzing is a widely-used technique for identifying software bugs. Despite their effectiveness, state-of-the-art fuzzers suffer from limitations in efficiency and effectiveness. Fuzzers based on random mutations are fast but struggle to generate high-quality inputs. In contrast, fuzzers based on symbolic execution produce quality inputs but lack execution speed. This paper proposes FlakJack, a novel hybrid fuzzer that patches the binary on the go to detect occluded bugs guarded by surface bugs. To dynamically overcome the challenge of patching binaries, the paper introduces multiple patching strategies based on the type of bug detected. The performance of FlakJack was evaluated on ten widely-used real-world binaries and one chaff dataset binary. The results indicate that many bugs found recently were already present in previous versions but were occluded by surface bugs. FlakJack's approach improved the bug-finding ability by patching surface bugs that usually guard occluded bugs, significantly reducing patching cycles. Despite its unbalanced approach compared to other coverage-guided fuzzers, FlakJack is fast, lightweight, and robust. False-Positives can be filtered out quickly, and the approach is practical in other parts of the target. The paper shows that the FlakJack approach can significantly improve fuzzing performance without relying on complex strategies.

ACKNOWLEDGMENTS

I want to express my deepest gratitude to my parents and sister for their unwavering support, love, and encouragement throughout my academic journey. Their belief in me has been a constant source of strength and motivation. I would also like to thank my dearest friends and family for their invaluable help and support. They have been my pillars of strength during difficult times.

I am deeply grateful to my thesis advisor, Dr. Tiffany Bao, for her guidance, mentorship, and inspiration to pursue research in Automated Bug Discovery. Dr. Bao's profound knowledge and expertise have shaped my research and academic career. I would also like to sincerely thank all other members of my thesis committee for their valuable insights and suggestions that helped me improve my work.

Finally, I am deeply grateful to all the lab members for the stimulating and enriching research environment that they have created, which has played a crucial role in shaping my academic and professional pursuits.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

# INTRODUCTION

Despite efforts to increase software resilience to security vulnerabilities, most software remains vulnerable to attacks. While numerous mitigation methods are in use today, thousands of security vulnerabilities have been discovered in the last 15 years *Vulnerability distribution of CVE security vulnerabilities by type* 2022, with memory corruption and control flow hijacking accounting for one-third of the vulnerabilities reported *Vulnerabilities by type* 2022. Discovering and fixing security vulnerabilities promptly is critical, which can cause substantial financial losses *Finance software bug causes $217m in investor losses* 2011. Given how the software continues to grow in size and complexity, code reviews and static analysis tends to be ineffective in fixing the security vulnerabilities *A brief introduction to fuzzing and why it's an important tool for developers* 2020. There is a need to automate this process. Dynamic analysis systems, for example, "fuzzers," monitor the native execution of an application to identify vulnerabilities. Fuzzing uncovers *How fuzzing can make your open-source project more secure and reliable* 2022 software programming errors that otherwise failed to be detected through manual analysis.

Security researchers have been actively designing new fuzzing techniques Liu et al. 2021 Chen and Chen 2018 Godefroid, Levin, Molnar, et al. 2008 Hsu et al. 2018 Peng, Shoshitaishvili, and Payer 2018. Prior work has focused on many aspects of the fuzzing process from seed processing J. Wang et al. 2017 Herrera et al. 2021 Chen et al. 2020, input mutation J. Wang et al. 2019 Lyu et al. 2019 Lyu et al. 2022 Chen and Chen 2018 Zhao et al. 2022 to directed fuzzing Canakci et al. 2021 T. Wang

et al. 2010 Ganesh, Leek, and Rinard 2009 Zhu et al. 2020. While these techniques successfully discover security vulnerabilities, most fuzzers stop at a crashing bug restricting the bug-finding ability of the fuzzer. Unless an alternative execution path is found, the bug, a surface bug, blocks the fuzzer from exercising the code deeper in the program resulting in the exploration of the same path triggering the same bug. For instance, a surface bug can cause the program that is being fuzzed to crash, which halts execution, preventing the fuzzer from exploring any of the code below the surface bug. Unless an alternative solution to find the root cause of the crashes is developed, the fuzzer treats all crashes generated from the root cause as unique.

Although finding crashes are good, potentially a surface bug, a crash prevents the fuzzer from progressing by limiting the input generation. The fuzzers take different paths to reach the crashing instruction resulting in duplicates. For instance, a buffer overflow of eight bytes can trigger 256 unique crashes.

Even though existing simultaneously as a surface bug, occluded bugs (bugs guarded by surface bugs), as shown in 1, are unreachable until the surface bug is patched as a part of the security development lifecycle. Thus, the occluded bugs may go unfixed for years after discovering the surface bug. For example, consider Android's stagefright vulnerability *Stagefright bug* 2016, which existed from Android 2.2 to 5.1 in the multimedia framework library. The bug was discovered in 2015 by Joshua Drake, even though existed since 2010, by running American Fuzzy Lop (AFL) using a manual approach of fuzz-crash-analyze-patch (FCAP).

Without using the FCAP approach, there can be a long delay between discovering a surface bug and the occluded bugs lurking behind it. Patching bugs enables the attacker to find occluded bugs which otherwise would have gone undetected. The delay between developing the security fix from reporting the bug to releasing the fix

on the repository is significant. This delay between reporting a bug and patching Portswigger 2022 further risks occluded bugs going unnoticed, leaving the users with vulnerable software. Furthermore, finding occluded bugs will significantly reduce the security lifecycle.

Manual FCAP is slow due to the time to triage a crash and develop a patch for it. Subsequently, for every crash found, manual analysis, triaging, and patching take time, especially when the program has multiple bugs. Due to this, finding occluded bugs are challenging. To speed up this process, we propose an automated way to do FCAP, FlakJack.

Developing a patch for a crash found is hard as the root cause needs to be determined on what led to the crash and insert the patch at the crashing instruction address. The binary must then be recompiled with the patch for the next fuzzing cycle. This manual FCAP can take a significant amount of time. We thus created FlakJack, a novel vulnerability excavation system designed to find occluded bugs. We developed multiple dynamic patching strategies that bypass surface bugs without altering the control flow, thus opening up the paths behind surface bugs and increasing the fuzzer's code coverage. Although FlakJack relies on fuzzing, FlakJack's dynamic patching is orthogonal to prior research; thus, FlakJack incorporates many state-of-the-art analysis techniques Stephens et al. 2016 and can be used widely.

In this paper, we describe the design of FlakJack and evaluate its performance on real-world binaries by discovering 85 vulnerabilities in three binutils binaries, 31 vulnerabilities in libtiff, 1 vulnerability in tcpreplay, and 2 in gpac. In addition, we show the effectiveness of FlakJack by performing a historical longitudinal analysis using FlakJack on older software versions. In the evaluation, we show that FlakJack

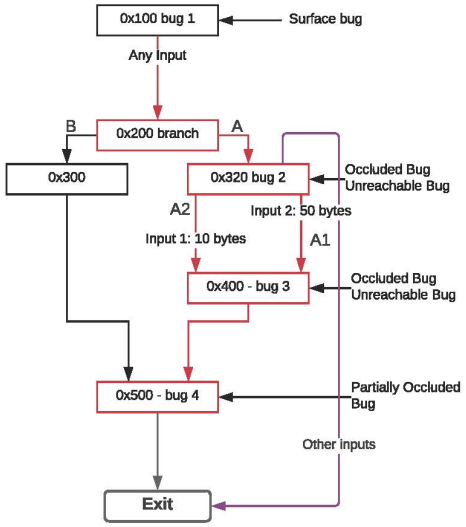


Figure 1. Occluded Bugs Along With the Surface Bug

dynamically patches surface bugs and discovers occluded bugs without the developer's official patch and with minimal false positives.

In summary, we make the following contributions:

- We propose a new automated method to improve the effectiveness of the coverage-guided fuzzer by using dynamic patching to enable the discovery of occluded bugs in real-world binaries.
- We designed and implemented a framework, FlakJack, to demonstrate this approach.
- We demonstrate the effectiveness of FlakJack by enabling and disabling each of the patching techniques in real-world binaries.

# Chapter 2

# BACKGROUND

## 2.1 Fuzzing

Fuzzing is the automated testing of vulnerabilities in software by feeding in randomly generated inputs to a program. Although fuzzing was introduced to test UNIX utilities *UNIX* 2020, fuzzing has evolved to become a standard practice in the security testing of applications. The basic premise of fuzz testing is to provide invalid, unexpected inputs into a system, a target binary, to identify failures.

The general workflow of a fuzzer involves several steps. First, a fuzzer requires a set of seeds, which it uses to construct a queue of test cases. The fuzzer selects a seed from the queue, randomly mutates, and inputs the generated test cases into the target program. This process repeats several times, and the fuzzer adds any exciting test cases that trigger new behavior into the input queue. The fuzzer then continues the process of generating inputs and testing.

One of the significant aspects of fuzzing is the ability to detect crashes. Once an input crashes the binary, the fuzzer uses the feedback to mutate the input and discover new paths. This process continues until the process's termination or the resource's exhaustion.

One of the most important driving factors of fuzzing is the seed used to generate test cases. A good seed corpus can significantly improve the efficiency and effectiveness of the fuzzing process. Seed inputs are from various sources, generated with prior

knowledge of the system, manually crafted inputs, or even previously generated test cases.

The next critical step in fuzzing is the mutational strategy for generating inputs. Different mutation strategies can be used based on the type of input generated, tested, and the behavior of the target program. For instance, mutational strategies are sufficient for testing integer inputs such as bit-flipping, whereas a more complex strategy is required for testing inputs such as file formats.

A fuzzer is evaluated based on the performance. Fuzzers that are slow usually take time to generate inputs which may not be able to generate suitable test cases within a specific time, whereas a fuzzer that can generate inputs much faster may not be able to generate inputs for complex cases.

## 2.2 Coverage-based Fuzzers

Although mutation-based fuzzers and dictionaries speed up the discovery of new paths in the binary, the aforementioned logic cannot solve complex constraints. To address the limitation, researchers tweaked the fuzzers to leverage coverage information as feedback to guide the fuzzing process. Since the aforementioned techniques are efficient and effective in accurately tracking the execution paths, and a fuzzer cannot find a vulnerability in a not-covered execution path, improving the coverage of the execution path is reasonable to enhance the fuzzing performance.

Several works employ adaptive strategies to improve coverage-based fuzzing. For instance, EcoFuzz Yue et al. 2020 focuses on adaptively adjusting the execution frequency of each seed test case on different programs. They deploy the Markov chain model and adversarial multi-armed bandit model to evaluate the potential of each

test case to trigger unique branching behaviors. After this, they allocate more time to mutate the promising test cases and vice versa. EMS Lyu et al. 2022 utilizes a Probablisitc byte orientation model to learn and reuse the efficient mutation strategies from the intra- and inter-trial fuzzing history with a fast execution speed that triggers unique paths and improves the mutation strategy. Other research in coverage-based fuzzing has been integrating mutation-based fuzzing with constraint-solving techniques, e.g., concolic execution Stephens et al. 2016. Such techniques utilize prior fuzzing history to identify unique inputs and solve path constraints. After tracing the inputs, the concolic execution utilizes the constraint-solving engine to identify the inputs that would force execution down new paths. On the other hand, Angora employs a gradient descent algorithm and several data tracking and analysis techniques to solve path constraints faster than concolic execution.

Maximizing bug discovery is a key subject of interest as it can gauge software robustness and also the effectiveness of the system. All these techniques find tons of crashes potentially terming as "unique" based on the crash addresses, although the root cause for these crashes can be the same bug. Fuzzer can spend more time generating inputs to explore the same state space as more unique crashes are found. All bugs crash at a particular program location. These crash sites, the address stored in the instruction pointer during the crash, serve as a bug identifier. Unfortunately, these crash sites are imprecise and can lead to bug misclassification (e.g., for use-after-free bugs, objects may be arbitrarily reused, triggering a broad set of "unique" crashes). Hence, crash sites both under and over-estimate bug counts. This leaves little incentive to triage crashes leaving it to maintainers to filter and fix bugs. With more crashes being reported, more time is spent on triaging the crashes and pruning duplicate bug reports, leaving maintainers with less time to fix bugs to improve the quality of the

software. For e.g., a buffer overflow of 8 bytes can generate 256 unique crashes. All these crashes are found to act as guards for the other potential bugs that are sitting behind them, which are missed out. Our approach tries to circumvent this by patching out the crashes.

Large fuzzing infrastructures *ClusterFuzz* 2023 *Announcing OSS-Fuzz: Continous fuzzing for open source software.* 2020, which run around the clock and automatically submit crash reports exacerbate this issue. For instance, as of February 2023, there are at least 1070 open bugs, with the oldest dating back to July 2020 *Syzbot* 2023. Solutions to these problems rely largely on the community to provide actionable analysis on their reports to filter out redundancies and duplicates Xu et al. 2017.

## 2.3 Binary Patching

Dynamic patching is a technique to modify a binary program while preserving its existing functionality. This approach involves the use of dynamic rewriting tools, such as Ramblr R. Wang et al. 2017, Pin *Pin - A Dynamic Binary Instrumentation Tool* 2020, and DynamoRIO Bruening 2004, to transform a binary program on the fly. These tools work by intercepting the execution of a program and rewriting its instructions to alter its behavior.

Some fuzzers, such as Flayer Drewry and Ormandy 2007 and TaintFuzz Bekrar et al. 2012, attempt to patch hard checks to make fuzzing easier by generating valid inputs that can bypass checks. This approach involves modifying certain program sections, such as checksums, to bypass security checks and facilitate input generation. However, this method can have unintended consequences, such as triggering false positives or altering the program's intended functionality.

In contrast, our approach focuses on patching only the crash instructions to make input generation more effective. When a crash is detected during the fuzzing process, we pause the execution and insert a patch at the location of the crashing instruction. This patch modifies the program behavior to prevent the crash from occurring, allowing the fuzzing process to continue without losing the generated inputs and executions.

By patching only the crash instructions, we avoid modifying other program parts that could result in unintended consequences. This approach allows us to target specific vulnerabilities without compromising the program's functionality. Additionally, by preserving the generated inputs and executions, we can continue the fuzzing process from the point of failure, allowing us to explore new paths and identify additional vulnerabilities.

## 2.4 Occluded bugs

We formally define surface bugs and occluded bugs and illustrate them with an example before discussing how to discover occluded bugs. Let $E$ denote the entry point of the program.

Surface bug: A bug $B$ at line $L$ is a surface bug if no bugs are present in all possible execution paths from $E$ to $L$.

Occluded bug: A bug $B_2$ at line $L_2$ is an occluded bug if, in all possible execution paths from $E$, there exists a line $L$ with a bug $B$ such that $L$ is executed before $L_2$.

Occluded bugs can be classified into two types: partially and fully occluded.

Partially occluded bug: A bug $B$ at line $L$ is a partially occluded bug if there exists at least one execution path $P_1$ from $E$ to $L$ where $B$ is a surface bug with a different execution path $P_2$ from $E$ to $L$ where $B$ is an occluded bug.

Fully occluded bug: A bug $B$ at line $L$ is a fully occluded bug if there exists no execution path from $E$ to $L$ where $B$ is a surface bug.

We now illustrate surface and occluded bugs with the example in listing B.1. For simplicity, we assume no bugs are present in any code executed before *function*. There are 3 bugs in listing B.1: a division by zero floating point exception at line 6, a null pointer dereference at line 9 and a stack-based buffer overflow at line 12. The division by zero exception at line 6 is a surface bug since there are no execution paths from the entry point of the program to line 6 with any bugs in them. The null pointer dereference at line 9 is a fully occluded bug since the division by zero exception at line 6 is present on all execution paths from entry point of the program through line 9. The stack-based buffer overflow at line 12 is a partially occluded bug. If the condition at line 4 is true, line 6 will be executed. Thus the division by zero error could be triggered before execution reaches line 12 and so the stack-based buffer overflow is occluded in this execution path. On the other hand, if the condition is false, line 12 is executed directly and thus it is a surface bug in this execution path.

# Chapter 3

# DESIGN

In this section, we show the framework, as shown in 2 of FlakJack and then present the design of the proposed patching techniques deployed in FlakJack.

## 3.1 Overview

FlakJack utilizes angr to generate the target binary's Control-Flow Graph (CFG). This CFG is then used to apply patches to the binary. Next, FlakJack starts to fuzz the target binary using the provided test cases. When the fuzzer identifies a crashing input, FlakJack pauses fuzzing and proceeds with crash triaging to determine the crash type.

Once the crash type is determined, FlakJack combines various techniques to identify the patch that needs to be applied. The determined patch is applied to the target binary to generate a patched binary, with which the fuzzing process resumes. This Fuzz-Crash-Analyze-Patch approach allows FlakJack to continuously identify

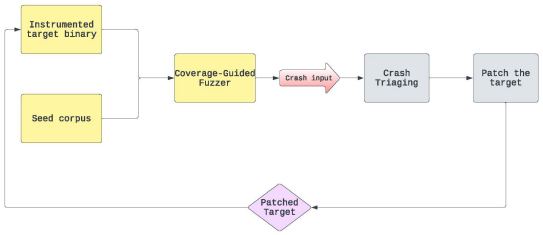


Figure 2. Overview of Flakjack System

and patch vulnerabilities within the target binary during fuzzing, resulting in finding deeper bugs.

Using angr to generate the CFG of the target binary is crucial in enabling FlakJack to identify potential patches. The CFG provides a comprehensive view of the binary's control flow, making it easier for FlakJack to analyze possible patches. Additionally, merging multiple techniques to identify the necessary patches is beneficial, increasing the likelihood of identifying the most effective solution.

## 3.2 Control-Flow Graph

The patching component of FlakJack requires a Control-Flow Graph (CFG) of the target binary to apply patches for any crashes that may occur effectively. This CFG determines the appropriate patch to apply by extracting the crashing instruction and corresponding registers for every crash. The CFG is also used to recreate the crashing instruction, which is then recreated in the patch that is ultimately applied. However, generating a CFG to apply patches for each crash can be time-consuming when many crashes are detected.

To address this issue, FlakJack has implemented a strategy to minimize the overhead associated with CFG generation. Specifically, FlakJack generates a CFG only after every tenth applied patch. This approach effectively reduces the time required to generate CFGs for patch application, allowing for efficient target binary patching. The selection of 10 as the arbitrarily set value for generating CFGs after every tenth patch balances between optimizing patch application time and ensuring that sufficient information is available to identify and apply the necessary patches accurately.

The use of CFGs in patching binary vulnerabilities is a common approach, as they provide a clear and concise representation of the target binary's control flow. By leveraging the CFG, FlakJack can identify and apply patches to specific instructions that are associated with crashes. The process of generating a CFG can be resource-intensive, however, so FlakJack's approach of generating CFGs after every tenth patch can be highly effective.

## 3.3 Crash Triaging

To determine the type of crash that has occurred, FlakJack employs various techniques. This section will provide a detailed account of the Crash Triaging process that FlakJack uses to identify the valid address of a crash.

To patch the crash, FlakJack requires the exact location of the crash, which is the instruction address. To accomplish this task, FlakJack uses the Crash Triaging process, which begins by creating a core dump. The core dump is generated by running the target binary against the crashing input generated by the fuzzer. Although the core dump identifies a crashing instruction, it is often not the source of the crash.

For instance, when a stack overflow occurs, the core dump will identify an address within the $\_\_stack\_check\_fail$ function, which is at the end of the function. Consequently, to identify the actual location of the crash, it is necessary to trace the calling function and determine the address of the instruction that led to the crash.

Using the core dump, FlakJack extracts the reported address of the crash. If the crashing instruction points to an unmapped memory region of the target binary, FlakJack employs the record-replay process to determine the actual location of the crash. In contrast, if the crashing instruction occurs in a mapped memory region of

the target binary, FlakJack passes the input that crashes the target binary and the target binary to the Precision Extractor component to refine further the instruction that led to the crash.

The record-replay process involves re-executing the target binary with the same input that caused the crash. However, RR instruments the binary during this re-execution to log all the memory accesses during the execution. The recorded trace is then replayed to identify the instruction that caused the crash.

On the other hand, Precision Extractor extracts features of interest from a program's execution. It does so by analyzing the execution trace of the program and identifying the instructions that led to the crash. Precision Extractor is used when the crashing instruction points to a mapped memory region of the target binary. In this case, Precision Extractor can quickly narrow down the instruction that led to the crash.

Once the valid location of the crash has been identified, FlakJack can determine the type of crash. The type of crash is determined by examining the system state when the crash occurred. The system state includes the contents of the CPU registers, the call stack, and the heap. By analyzing the system state, FlakJack can determine the root cause of the crash.

### 3.3.1 Record and Replay

When an overflow overwrites the instruction pointer, automatically identifying the address of the instruction that led to the crash is difficult. To solve this problem, FlakJack utilizes Record and Replay (RR), a tool developed by Mozilla *Mozilla:Record*

*and Replay* 2021 that enables the recording and debugging of program execution failures.

RR records all non-deterministic inputs, such as signals, timer interrupts, and file I/O, during program execution. When the recorded execution is replayed, RR simulates the inputs in the same order and timing as the original execution. This ensures that the replayed execution follows the same execution path as the original execution. During the replayed execution, RR also monitors the program's memory accesses and system calls. If the program makes any non-deterministic system calls, RR blocks the call and simulates the system call's return value based on the recorded input.

FlakJack leverages RR to record the execution of the target binary with the crashing input. The recorded execution is then replayed until the point of the crash. The execution is then replayed in reverse to identify the instruction that overwrote the instruction pointer during the crash. This enables FlakJack to pinpoint the instruction address that caused the overwrite, which is then passed on to the patching component for further action.

### 3.3.2 Precision Extractor

The Crash Triaging component retrieves the address of the instruction that led to the crash, and further analysis is conducted using the Precision Extractor component. This analysis can be challenging, as the address may point inside a library function, necessitating further analysis that requires the invocation of the rational extractor.

The Precision Extractor component is designed to identify and extract the relevant information from the crash report, which can help identify the crash's root cause.

The component uses various techniques, including debugging, to gather information about the crash. This information is then analyzed to determine the specific cause of the crash. In cases where the address points inside a library function, the rational extractor is invoked to analyze the crash further.

### 3.3.3 Rational Extractor

Crashes can occur due to various reasons, including the passing of corrupt arguments in library function calls. However, patching library functions can result in unintended consequences, as multiple functions can make multiple calls to these functions, triggering the patch at any call and causing unwanted behavior. Additionally, 128-bit registers in the crashing instruction can further complicate patching inside library functions. Therefore, to avoid tampering with library functions, a rational extractor is used.

The rational extractor component extracts and analyzes the relevant code that led to the crash. When a crash occurs inside a library function, call the rational extractor invokes gdb to run the target software along with the crashing input. This allows for the call stack to be determined, providing the address of the library call from the parent function. However, in some cases, the backtrace may be corrupted, resulting in the inability to retrieve the parent function's address. In such instances, this component generates a trace of the execution using qemu, which provides a log file containing the trace of the blocks executed until the crash instruction.

By leveraging qemu logging, the target binary is run against the input generated by the fuzzer, and the log file obtained from the qemu logging is analyzed to determine the execution trace until the crashing instruction. The trace obtained from the gdb

backtrace, along with the qemu trace, is then used to determine the address of the instruction right before the call to the library function. This address is passed on to the patching component of FlakJack. This prevents unintended consequences of patching inside the library function and ensures that the stability and reliability of the software system are maintained.

## 3.4 Dynamic Patching

Determining the patch for a crash found by FlakJack in the target binary is determined by this component. This component begins by identifying the type of crash by using multiple methods since several causes exist. Applying a single generic patch for all crashes triggered is not ideal and can affect the program semantics.

Upon obtaining the address of the crashing instruction, the component proceeds to apply patches. However, as mentioned earlier, it is not feasible to apply a generic patch, as various factors can trigger crashes, and applying a generic patch may affect the behavior and control flow of the target binary. We have developed several patching strategies that fix specific crashes to address this problem. These patching strategies enable the fuzzer to bypass the problematic code and allow the target binary to continue executing without affecting the program's semantics.

FlakJack has developed five distinct patching strategies to address different types of crashes. By using multiple patching strategies, we can ensure that the fuzzer can handle various crashes that may occur during the fuzzing process. The patching strategies enable to facilitate the continued execution of the target binary.

Following are the different types of patching strategies deployed by FlakJack.

- Reckon patch: The Reckon Patch addresses the issue of floating-point exceptions

that arise when a program attempts to perform an impossible operation with a floating-point number, such as dividing by zero. When the target program crashes, FlakJack identifies the operation that caused the crash using the control flow graph (CFG) generated before fuzzing. Once the cause of the crash is determined, the Reckon Patch applies to the instruction that led to the crash. The Reckon Patch behaves in such a way that it activates only when a particular register value is corrupt. It rewrites the crashing instruction and replaces it with an assembly that first compares the register value against 0. If the value is not 0, the execution of the program remains the same. However, if the operand register value is 0, the patch replaces the value of the register with a value stored in memory that holds 8 bytes from the input. The Bridgehead Patch performs this memory storage. The Bridgehead Patch works in tandem with the Reckon Patch to enable the replacement of a corrupt value at the crashing instruction. It is designed to provide input only when the program is in a specific state. In this case, the Bridgehead Patch provides input when the register value is 0. By rewriting the crashing instruction and inserting additional code into the program, these patches allow the program to continue running even when an impossible operation with a floating-point number occurs. Consider the disassembly of an x86-64 binary at the crashing instruction as listing B.3. In this case, r8 with zero will lead to a crash halting the execution of the target program. FlakJack rewrites the crashing instruction and replaces it with the assembly as on listing C.2. The patch essentially behaves in such a way that it is activated only when the value of the register is corrupt. Initially, as in line 1, r8 is compared against 0, and if the value of the register is not 0, then execute the original instruction. On the other hand, if the value of the operand register is 0, the value of the

register is replaced with a value from memory, which stores 8 bytes from the input.

- Legion patch: The Legion Patch is a technique deployed by FlakJack in situations where an instruction involving multiple registers or arithmetic operations results in memory addresses that leads to a crash when the memory address points to an unmapped memory region. When such an event occurs, the patch must verify whether the values of the registers lie within the mapped memory area. The patch is activated by FlakJack to assess the values of registers involved in an arithmetic operation, confirming whether the resulting value resides within the mapped memory range. If the value of the result falls outside the mapped memory region, the patch replaces the value of the register with the address of a newly created page holding 8 bytes of input. FlakJack has developed the patch to address the issue of execution halting due to a segmentation fault caused by accessing an invalid memory address. For example, consider the scenario in Listing B.2, where the value of register r12 holds the result of an arithmetic operation between registers rdx and rcx. If the result of the operation points to an unmapped memory region, the instruction causes a segmentation fault, halting the execution of the target binary. To mitigate such an occurrence, the Legion Patch follows a specific algorithm. Initially, the value of the register rdx is compared to the lower and upper mapped memory region. During execution, if the patch detects a corrupted value in the register, the patch replaces it with the address of the new page. The exact process is used to check the value of the other register, rcx. The Bridgehead patch, further explained later, stores 8 bytes for every patch inserted from the input at the beginning of a newly created page. The Legion Patch's primary objective is to ensure that the arithmetic

operation executed does not result in invalid memory access that could cause the program's termination. As a result, the patch verifies the registers' values and replaces any corrupted value with the address of a newly created page. The algorithm used in the patch is critical in maintaining the integrity of the program's execution and preventing unnecessary termination.

- Colony patch: The Colony Patch is a patch used by FlakJack to address crashes that involve the value of a register at the crashing instruction to control the program execution within the same function. To determine whether the register's value is used in compare or test instructions, FlakJack leverages angr to conduct a symbolic exploration. This process involves assigning a symbolic value to the source register in the crashing instruction and analyzing the register's use in subsequent instructions. If the analysis reveals that the register's value controls the program's flow, the Colony Patch is applied. The patch ensures the target binary can continue executing by taking multiple paths in the compare or test instruction. To ensure this, the patch replaces the crashing instruction with a div operation that randomizes the value of the register when it is corrupted. The value used for the div operation is obtained from the input stored at the beginning using the Bridgehead patch. This approach ensures that the patch effectively mitigates the crash and enables the fuzzer to continue testing the target binary.
- Fountainhead patch: The Fountainhead Patch is a technique employed by FlakJack to address crashes within library functions, typically caused by corrupted arguments passed to the function. In such cases, FlakJack utilizes multiple techniques to identify the address of the call to the library function. The Fountainhead Patch is inserted before the call to the library function to ensure that

the arguments passed to the function are not corrupted. To accomplish this, the Fountainhead Patch replaces the value of any corrupted argument with the address of the page newly created by the Bridgehead Patch. This approach ensures that the library function is executed with valid arguments and avoids crashes caused by corrupted arguments. One key advantage of the Fountainhead Patch is that it is inserted before the library function calls. Applying a patch this way is essential because many crashing instructions inside library functions can have 128-bit registers (xmm) for which patching is not feasible. Patching these registers can have unintended consequences for every library call in the target binary. By inserting the patch before the function call, FlakJack avoids the need to patch the registers, instead assuming that the culprit is one of the arguments and focusing the patch accordingly. For example, consider a case where the source address in a memcpy call is corrupted and points to an invalid memory region. The Fountainhead Patch overwrites the corrupted source address with the new page address that holds the value from the input fed by the fuzzer. This patch ensures that the library function can execute with valid arguments and avoids crashes caused by corrupted arguments. Also, the Fountainhead Patch is only invoked when the argument values are corrupted, thereby avoiding any crashes caused by such corruption. This approach mitigates crashes within library functions and enables FlakJack to continue testing the target binary with minimal interruptions.

- Bridgehead patch: The Bridgehead patch used by FlakJack is the crucial patch compared to other patches. This patch has two functionalities, unlike other patches. First, the patch adds a new page with a static address to the target binary. Secondly, the patch inserts a code block at the entry point of the target

binary as shown in listing C.1. For a patch, as mentioned earlier, to be applied, FlakJack invokes the Bridgehead patch, along with other patches. For instance, if two patches are applied, one Colony and one Fountainhead patch, the value of n as shown in C.1 will be 2. In that case, the code block inserted at the entry point copies the first 16 bytes of the input to the new page. The address of the new page is static (0x700000), and hence FlakJack can determine the address of the bytes stored from the input in order and for corresponding patches. In the example, the first patch applied is the Colony patch; hence, the first 8 bytes, from 0x700000 to 0x700007, in the new page will always be used for the same patch when required; that is when the crashing instruction requires the input value. Similarly, the next 8 bytes, from 0x700008 to 0x700010, are reserved for the Fountainhead patch. If the patch requires an address to be moved into the register to continue the execution without crashing, the address of the new page created replaces the corrupt value of the register. The new page created is 0x1000 in size and can hold inputs for approximately 512 crashes for the target, ensuring that FlakJack can handle many crashes. This patch acts as the foundation for all the aforementioned patching techniques, as it glues them together by helping other patches to utilize the values stored in new pages to prevent crashes.

## 3.5 Dependency Exploration

FlakJack uses Symbolic exploration leveraged from angr. Before fuzzing starts, FlakJack creates a static Control Flow Graph(CFG), which uses static analysis to generate a CFG, which is then used for the patching component of FlakJack. CFG

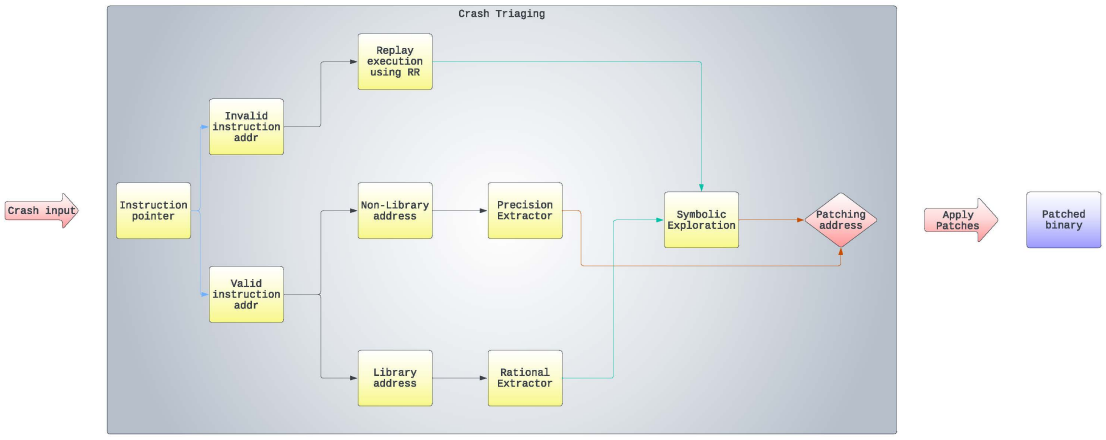


Figure 3. Implementation of Crash Triaging Component

recovery performs a static control flow and functional recovery. Starting with the entry point, the basic blocks are lifted to VEX IR, and subsequently, all exits are collected. In the event of a function call, the destination block is considered the start of a new function. FlakJack requires all this information to determine what patch needs to be applied and extract registers at crashing instruction. We also modified Patcherex to take in a pre-generated CFG while loading a binary into it. We only generate the CFG once at the beginning to reduce the overhead for CFG creation for every run after patching the binary.

When a crash input is found, FlakJack does symbolic exploration to find any dependencies of the destination register on any future compare, call or test instructions to determine which patch to be applied, limited to the current function. A dependency patch is inserted at the crashing instruction if any such dependencies are found. A blank state is created at the crash's address, leaving most data uninitialized. The exploration is limited to only the current function, where the crashing instruction deals with the state explosion problem, mainly a bottleneck for symbolic exploration.

## 3.6 Implementation

We leverage angr to generate the CFG of the target binary. Once a patch is applied, new blocks are added for which the CFG differs from the unpatched target binary. Since CFG generation takes time, we generate a new CFG after every 10th patch is applied, wherein 10 is an arbitrarily fixed number. We use Patcherex *Patcherex* 2022 to apply patches to the target binary when a crash is found. Here, we use AFLplusplus Fioraldi et al. 2020 as the base fuzzer on which FlakJack is implemented to fuzz the target. FlakJack relies on rr to replay the execution to fetch the address, which leads to the crash of the target, and pwntools *CTF framework and exploit development library* 2022 along with qemu *A generic and open source machine emulator and virtualizer* 2023 and gdb *GDB: The GNU Project Debugger* 2022 to extract the instruction address from the core dump. FlakJack also exposes Python APIs so users can easily extend the dynamic patching to other fuzzers. The current implementation of FlakJack supports 32-bit and 64-bit Linux binaries and will be open-sourced.

# Chapter 4

# EVALUATION

In this section, we evaluate the fuzzing performance of FlakJack following the guidelines in Klees et al. 2018. To determine FlakJack's bug-finding effectiveness, we evaluated seven real-world binaries. We also show the robustness of FlakJack by adding patching onto the existing fuzzer, AFLplusplus, to improve the code coverage.

In this section, we are looking to evaluate the following research questions.

- Does an Occluded bug exist? If so, does dynamic patching enable the fuzzer to find occluded bugs?
- How effective are each of the patching techniques?

The experiments were run on docker containers in which each container runs Debian 10 and is equipped with an Intel Xeon processor and 251 GB of memory.

## 4.1 ABLATION STUDY

This experiment shows the impact of different patching techniques on the overall effectiveness of FlakJack. The study's primary objective was to contrast each patching technique's contributions by selectively deactivating or activating them, thereby facilitating a comprehensive evaluation of their impact.

To conduct the ablation study, each patching technique incorporated within the FlakJack system was systematically disabled one at a time while maintaining the integrity of the remaining components. The experiments were run by modifying the configuration settings of FlakJack to exclude specific patching techniques. Using

this methodology, the study aimed to isolate the effects of each patching technique, permitting a focused assessment of their respective contributions to the overall FCAP process.

Throughout the ablation study, experiments were run on ten real-world binaries. The impact of turning off each patching technique was analyzed regarding FlakJack's efficacy in identifying and patching vulnerabilities. The outcomes of these experiments were recorded and summarized in Table A.

By comparing the results obtained from experiments conducted with the enabled and disabled patches, the experiment aimed to derive insights into each patching technique's strengths and weaknesses. The metrics considered for evaluation included vulnerability detection and the discovery of occluded bugs. This analysis enabled a comprehensive understanding of the impact of each patching technique on FlakJack's effectiveness in identifying and mitigating software vulnerabilities.

### 4.1.1 Real-World Binaries

This study aimed to evaluate the effectiveness of FlakJack in detecting previously unknown occluded and partially occluded bugs in real-world binaries, specifically, the binary tiffcp, version 4.0.0, released in 2011 and continued until the latest version, 4.5. The results were analyzed and are presented in Table 1 and figure 4. The study employed a case study approach where FlakJack was used to fuzz the binary tiffcp.

During the study, several occluded and partially occluded bugs were discovered in the earlier binary tiffcp. Of particular interest was the discovery of a bug in version 4.0.2 that remained present in the latest version but was previously unknown to developers. A surface bug concealed this occluded bug, making it difficult to detect

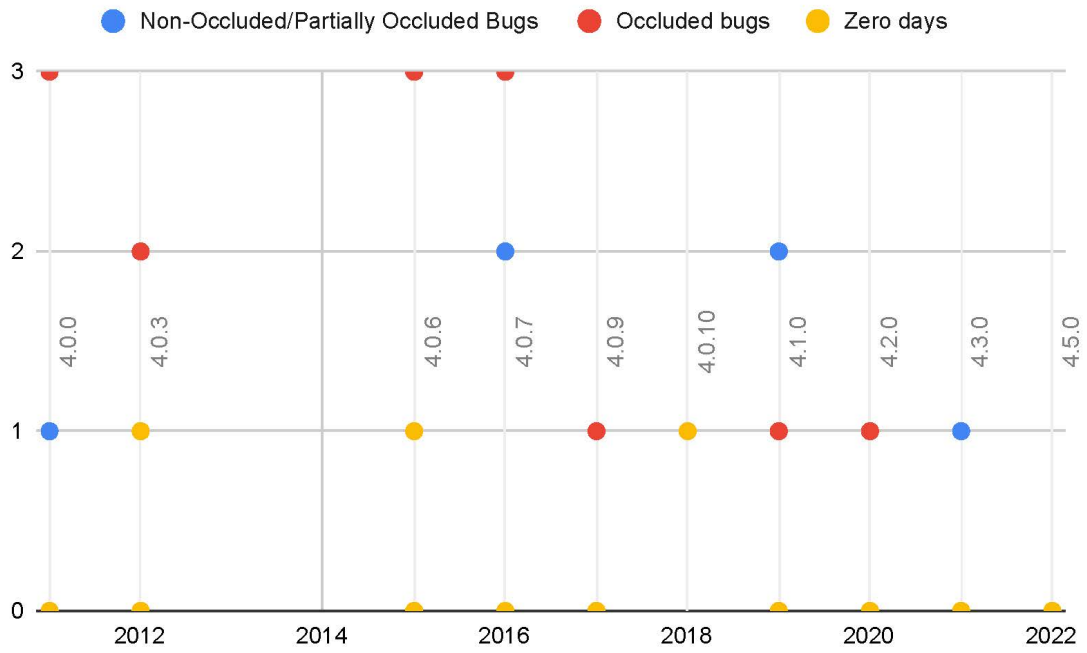


Figure 4. Case Study on Tiffcp

using conventional fuzzing techniques. However, Flakjack's patching strategy enabled the discovery of the occluded bug.

FlakJack's success in detecting the occluded bug high- lights the effectiveness of its patching strategy in uncovering previously unknown bugs. This approach can potentially assist developers in improving their software's overall security and reliability. Additionally, the discovery of multiple occluded and partially occluded bugs in the earlier versions of tiffcp demonstrates the need for more sophisticated fuzzing techniques that can detect such bugs.

# Chapter 5

# LIMITATION

Flakjack is a dynamic binary analysis tool that applies patches to binaries to mitigate crashes. However, in specific scenarios, Flakjack may encounter some limitations discussed in this section.

Firstly, in some cases, the root-cause identification of Flakjack may fail, resulting in an imprecise patch applied at a different location than required. This imprecision can affect the behavior of the target binary. Additionally, when fetching the crash instruction address is difficult, such as when a corrupted backtrace is encountered, Flakjack may be unable to apply patches. If Flakjack cannot obtain the crashing address, the crashing input is discarded from the fuzzing history and is stored at a separate location. Secondly, Flakjack rewrites the crashing instruction while applying patches. However, rewriting instructions that contain certain operations, such as movaps and movups, is currently not supported by Patcherex. Therefore, crashes that involve these operations are usually not patched. Thirdly, when applying patches using Patcherex, a limitation exists where the new patch will not be applied if less than five bytes are available in the same basic block. Fourthly, while applying the Reckon patch, there is a slight chance that the value being restored from the newly created page is zero, which leads to a crash inside the patch. Although the probability of such an event is very low, it can result in a false positive. Lastly, when an arithmetic operation of registers at the site of a Legion patch results in a crash, Flakjack reports it as a true positive when it is a false positive. This is an ongoing problem, and

automated techniques are currently insufficient to address it. Modifying the patch to check the registers' addresses during each execution may be a potential solution.

Moreover, Patcherex takes approximately 0.1 to 0.3 seconds to apply a patch in a target binary. When the number of patches is significant, FlakJack loses time generating a patched binary to resume fuzzing. While this can be an edgy case, the overhead for applying patches still exists, which we believe is an engineering problem that can be solved in the future.

# Chapter 6

# DISCUSSION & CONCLUSION

## 6.1 Discussion

FlakJack is a dynamic binary analysis tool that applies patches to binaries to mitigate crashes. However, in specific scenarios, FlakJack may encounter some limitations discussed in this section.

One issue is that root-cause identification in FlakJack may fail. This could result in an incorrect patch being applied or a patch being applied at an incorrect location. This could result in false positive crashes being generated in future fuzzing iterations. Additionally, if FlakJack cannot determine the crash instruction's address for various reasons (eg: the call backtrace is corrupted), FlakJack may be unable to apply patches. If FlakJack cannot obtain the crashing address, the crashing input is discarded from the fuzzing history and is stored at a separate location. Secondly, FlakJack relies on Patcherex for applying patches. However, Patcherex does not support the entire machine instruction set. Therefore, crashes that involve these operations are usually not patched. Thirdly, when applying patches using Patcherex, a limitation exists where the new patch will not be applied if less than five bytes are available in the same basic block. Fourthly, while applying the Reckon patch, there is a slight chance that the value being restored from the newly created page is zero, which leads to a false positive crash inside the patch. Lastly, when an arithmetic operation of registers at the site of a Legion patch results in a crash, FlakJack reports it as a true positive when it is a false positive. This is an ongoing problem, and automated techniques

are currently insufficient to address it. Modifying the patch to check the registers' addresses during each execution may be a potential solution. Moreover, Patcherex takes approximately 0.1 to 0.3 seconds to apply a patch in a target binary. When the number of patches is significant, FlakJack loses time generating a patched binary to resume fuzzing. While this can be an edge case, the overhead for applying patches still exists, which we believe is an engineering problem that can be solved in the future.

## 6.2 Conclusion

This thesis presented and evaluated methods based on patching to improve fuzzing. The results demonstrate that patching out surface bugs that typically guard deep-lying occluded bugs can significantly improve bug-finding ability and reduce patching cycles. Although our approach is not as well-balanced as other fuzzing techniques, we believe it upholds the core strengths of coverage-guided fuzzer: fast, lightweight, and robust. Despite the possibility of false positives resulting from patches, they can be quickly filtered out, making our approach practical and applicable in other parts of the target. The work presented in this paper highlights the potential to improve fuzzing performance without the need for complex strategies significantly.

# REFERENCES


*A brief introduction to fuzzing and why it's an important tool for developers.* 2020. https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-\an-important-tool-for-developers/.

*A generic and open source machine emulator and virtualizer.* 2023. https://www.qemu.org.

*Announcing OSS-Fuzz: Continous fuzzing for open source software.* 2020. https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

Bekrar, Sofia, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2012. "A taint based approach for smart fuzzing." In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation,* 818–825. IEEE.

Bruening, Derek. 2004. "Efficient, transparent, and comprehensive runtime code manipulation."

Canakci, Sadullah, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. "DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing." In *2021 58th ACM/IEEE Design Automation Conference (DAC),* 529–534. https://doi.org/10.1109/DAC18074.2021.9586289.

Chen, Peng, and Hao Chen. 2018. "Angora: Efficient fuzzing by principled search." In *2018 IEEE Symposium on Security and Privacy (SP),* 711–725. IEEE.

Chen, Yaohui, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. "{MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing." In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020),* 77–92.

*ClusterFuzz.* 2023. https://google.github.io/clusterfuzz/.

*CTF framework and exploit development library.* 2022. https://github.com/Gallopsled/pwntools.

Drewry, Will, and Tavis Ormandy. 2007. "Flayer: Exposing application internals."

*Finance software bug causes $217m in investor losses.* 2011. https://www.theregister.com/2011/09/22/software_bug_fine/.

Fioraldi, Andrea, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. "AFL++: Combining incremental steps of fuzzing research." In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20).*

Ganesh, Vijay, Tim Leek, and Martin Rinard. 2009. "Taint-based directed whitebox fuzzing." In *2009 IEEE 31st International Conference on Software Engineering,* 474–484. IEEE.

*GDB: The GNU Project Debugger.* 2022. https://www.sourceware.org/gdb/.

Godefroid, Patrice, Michael Y Levin, David A Molnar, et al. 2008. "Automated whitebox fuzz testing." In *NDSS,* 8:151–166.

Herrera, Adrian, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. "Seed selection for successful fuzzing." In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis,* 230–243.

*How fuzzing can make your open-source project more secure and reliable.* 2022. https://developer.ibm.com/blogs/how-fuzzing-can-make-your-\open-source-project-more-secure-and-reliable/.

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

Klees, George, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. "Evaluating fuzz testing." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security,* 2123–2138.

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

Lyu, Chenyang, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. "{MOPT}: Optimized mutation scheduling for fuzzers." In *28th USENIX Security Symposium (USENIX Security 19),* 1949–1966.

Lyu, Chenyang, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 2022. "EMS: History-Driven Mutation for Coverage-based

Fuzzing." In *29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss.*

*Mozilla:Record and Replay.* 2021. https://github.com/rr-debugger/rr.

*Patcherex.* 2022. https://github.com/angr/patcherex.

Peng, Hui, Yan Shoshitaishvili, and Mathias Payer. 2018. "T-Fuzz: fuzzing by program transformation." In *2018 IEEE Symposium on Security and Privacy (SP),* 697–710. IEEE.

*Pin - A Dynamic Binary Instrumentation Tool.* 2020. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

Portswigger. 2022. *Weaknesses in open source patch process.* https://portswigger.net/daily-swig/lagging-behind-new-study-highlights-weaknesses-in-\open-source-patch-process.

*Stagefright bug.* 2016. https://en.wikipedia.org/wiki/Stagefright_(bug).

Stephens, Nick, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In *NDSS,* 16:1–16. 2016.

*Syzbot.* 2023. https://syzkaller.appspot.com/upstream.

*UNIX.* 2020. https://dl.acm.org/doi/pdf/10.1145/96267.96279.

*Vulnerabilities by type.* 2022. https://www.cvedetails.com/vulnerabilities-by-types.php.

*Vulnerability distribution of CVE security vulnerabilities by type.* 2022. https://www.cvedetails.com/browse-by-date.php.

Wang, Junjie, Bihuan Chen, Lei Wei, and Yang Liu. 2017. "Skyfire: Data-driven seed generation for fuzzing." In *2017 IEEE Symposium on Security and Privacy (SP),* 579–594. IEEE.

———. 2019. "Superion: Grammar-aware greybox fuzzing." In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE),* 724–735. IEEE.

Wang, Ruoyu, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. "Ramblr: Making Reassembly Great Again." In *NDSS.*

Wang, Tielei, Tao Wei, Guofei Gu, and Wei Zou. 2010. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection." In *2010 IEEE Symposium on Security and Privacy,* 497–512. IEEE.

Xu, Jun, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. "Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts." In *USENIX Security Symposium,* 17–32.

Yue, Tai, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. "{EcoFuzz}: Adaptive {Energy-Saving} Greybox Fuzzing as a Variant of the Adversarial {Multi-Armed} Bandit." In *29th USENIX Security Symposium (USENIX Security 20),* 2307–2324.

Zhao, Xiaoqi, Haipeng Qu, Jianliang Xu, Shuo Li, and Gai-Ge Wang. 2022. "AMSFuzz: An adaptive mutation schedule for fuzzing." *Expert Systems with Applications* 208:118162.

Zhu, Xiaogang, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Camtepe Seyit, and Yang Xiang. 2020. "Defuzz: Deep learning guided directed fuzzing." *arXiv preprint arXiv:2010.12149.*

# APPENDIX A

# EXPERIMENT RESULTS

Table 1. Case Study on `Tiffcp`

| **Version** | **Partially Occluded Bugs** | **Fully Occluded Bugs** | **Zero Days** |
|---|---|---|---|
| 4.0.0 | 1 | 3 | 0 |
| 4.0.1 | 1 | 1 | 0 |
| 4.0.2 | 1 | 1 | 1 |
| 4.0.3 | 1 | 2 | 1 |
| 4.0.4 | 1 | 3 | 1 |
| 4.0.5 | 1 | 0 | 0 |
| 4.0.6 | 1 | 1 | 1 |
| 4.0.7 | 2 | 3 | 0 |
| 4.0.8 | 1 | 1 | 0 |
| 4.0.9 | 1 | 0 | 0 |
| 4.0.10 | 1 | 1 | 1 |
| 4.1 | 2 | 1 | 0 |
| 4.2 | 1 | 1 | 0 |
| 4.3 | 1 | 0 | 0 |
| 4.4 | 0 | 0 | 0 |
| 4.5 | 0 | 0 | 0 |

| Program | Colony | | Fountainhead | | Legion | | Reckon | |
|---|---|---|---|---|---|---|---|---|
| | Enable | Disable | Enable | Disable | Enable | Disable | Enable | Disable |
| readelf | 1 | 2 | 1 | 4 | 2 | 1 | 1 | 2 |
| objdump | 3 | 9 | 2 | 11 | 2 | 6 | 2 | 3 |
| nm | 7 | 14 | 2 | 17 | 2 | 13 | 9 | 11 |
| tiffcp | 6 | 14 | 9 | 6 | 6 | 14 | 3 | 7 |
| size | 2 | 2 | 3 | 9 | 2 | 7 | 1 | 4 |
| tcpreplay | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| gpac | 3 | 2 | 2 | 2 | 2 | 1 | 2 | 1 |
| tiffset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| tiff2pdf | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| tiffinfo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 2. Occluded Bugs Found by Flakjack with Each Technique Enabled and Disabled

# APPENDIX B

# RAW CODE LISTING

Listing B.1. Example code showcasing surface, partially occluded & fully occluded bugs

```
int function(int command) {
  int num1, num2, num3, ret_val;
  char buffer[1024];
  if (command & 3 == 3) {
    scanf("%d %d", &num1, &num2);
    num3 = num1 / num2;    // Surface bug (#6)
    if (num3 < 4) {
      addr = g_addr_table * (num3 - num2 + 1);
      ret_val = *addr; // Fully occluded bug (#9)
    }
  }
  gets(buffer); // Partially occluded bug (#12)
  if(strncmp(buffer, "admin", 5) == 0) {
    ret_val = 1;
  }
  return ret_val;
}
```

Listing B.2. Instruction Involving Multiple Registers

```
mov r12, [rdx+rcx*3]
```

Listing B.3. R8d Having 0 Causes SIGFPE

```
; rcx is set to 0
movsxd rdx, edx
mov r8d, rcx
div r8d
```

# APPENDIX C

## PATCH CODE INSERTED FOR DIFFERENT CRASHES

Listing C.1. Patch Code for Bridgehead Patch

```
;address of the new page where input
;bytes are stored
mov r13, {page_addr}
;counter value is set to determine
;the memory location of the input stored
mov r14, {counter * 8}
mov r13, 0x07000300 ; address of from the new page
mov [0x07000220], r14
add r14, 4
l0:
    mov BYTE [r13], 0
    add r13, 1
    sub r14, 1
    cmp r14, 0
    jne l0
xor r13, r13

;get first argument
;which is the file name
;where we have to read n bytes
xor r14, r14
add r14, 10h
add r14, counter*8
add rsp, r14
pop rdi

;open the file
mov rax, 2
mov rsi, 0
syscall
mov [0x07000200], rax

;re-initiate stack to original structure
xor r14, r14
add r14, 0x18
add r14, counter*8
sub rsp, r14

;reset every register
xor rdx, rdx
```

```
xor rsi, rsi
xor rdi, rdi
xor r14, r14
xor r13, r13
xor rax, rax
xor r11, r11
```

Listing C.2. Patch Inserted for SIGFPE Reckon Patch

```
cmp r8d, 0
jle label1
div r8d
jmp end
label1:
   ;value is retained from the
   ;newly created page address
   ;where certain bytes of
   ;input are stored
  mov r8d, page_addr
   div r8d
end:
   nop
```

Listing C.3. Patch Code for Camouflage Patch

```
cmp source_reg, mapped_mem_region
jle sec1
cmp source_reg, mapped_mem_region
jge sec1
original_instruction
jmp end_patch
sec1:
    cmp source_reg, mapped_mem_region
    jle sec2
    cmp source_reg, mapped_mem_region
    jge sec2
    original_instruction
    jmp end_patch
sec2:
    push rdx
    push rcx
    push rsi
    ;counter value is set to determine
    ;the memory location of the input stored
    mov rcx, counter
    cmp rcx, mapped_mem_region
    jl zero_check
    mov rcx, -1
    ;address of the newly created page
    mov rsi, counter_addr
    mov rsi, rcx
    jmp normal_path
zero_check:
    cmp rcx,0
    je reset
reset:
    mov rcx,-1
    mov rsi, counter_addr
    mov rsi, rcx
normal_path:
    mov rdx, address
    mov rsi, [counter_addr]
    add rsi, 1
```