Fuzz to the Future: Uncovering Occluded Future Vulnerabilities via Robust Fuzzing

Arvind S Raj
Arizona State University
arvindsraj@asu.edu

Wil Gibbs
Arizona State University
wfgibbs@asu.edu

Fangzhou Dong
Arizona State University
fdong12@asu.edu

Jayakrishna Menon Vadayath
Arizona State University
jvadayat@asu.edu

Michael Tompkins
Arizona State University
mctompk1@asu.edu

Steven Wirsz
Arizona State University
swirsz@asu.edu

Yibo Liu
Arizona State University
yiboliu@asu.edu

Zhenghao Hu
New York University
huzh@nyu.edu

Chang Zhu
Arizona State University
czhu62@asu.edu

Gokulkrishna Praveen Menon
Arizona State University
gpraveen@asu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Adam Doupé
Arizona State University
doupe@asu.edu

Ruoyu Wang
Arizona State University
fishw@asu.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Tiffany Bao
Arizona State University
tbao@asu.edu

ABSTRACT

The security landscape of software systems has witnessed considerable advancements through dynamic testing methodologies, especially fuzzing. Traditionally, fuzzing involves a sequential, cyclic process where software is tested to identify crashes. These crashes are then triaged and patched, leading to subsequent cycles that uncover further vulnerabilities. While effective, this method is not efficient as each cycle potentially reveals new issues previously obscured by earlier crashes, thus resulting in vulnerabilities being discovered sequentially.

In this paper, we present a solution to identify *occluded future vulnerabilities* — vulnerabilities that are hard or impossible to trigger due to current vulnerabilities occluding the triggering path. We introduce robust fuzzing, a novel technique that enables fuzzers probe beyond the immediate crash location and uncover new vulnerabilities or variants of known ones. We implemented robust fuzzing in FlakJack, a pioneering fuzzing add-on that leverages binary patching to proactively identify occluded future vulnerabilities hidden behind current crashes. By enabling fuzzers to bypass immediate crash points and delve deeper into the software, FlakJack not only accelerates the vulnerability discovery process but also significantly enhances the efficacy of software testing. With the help of FlakJack, we found 28 new vulnerabilities in projects that have been extensively tested through the OSS-Fuzz project. This approach promises a transformative shift in how vulnerabilities are identified and managed, aiming to shorten the time span of vulnerability discovery over the long term.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Software Security, Binary Analysis, Fuzzing

ACM Reference Format:
Arvind S Raj, Wil Gibbs, Fangzhou Dong, Jayakrishna Menon Vadayath, Michael Tompkins, Steven Wirsz, Yibo Liu, Zhenghao Hu, Chang Zhu, Gokulkrishna Praveen Menon, Brendan Dolan-Gavitt, Adam Doupé, Ruoyu Wang, Yan Shoshitaishvili, and Tiffany Bao. 2024. *Fuzz to the Future:* Uncovering Occluded Future Vulnerabilities via Robust Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690278

1 INTRODUCTION

The detection of vulnerabilities, once the domain of manual code auditing by expert human practitioners, has become increasingly automated over the years. This automation has been greatly powered by the rise of *fuzzing*, a stochastic dynamic analysis that has become a premier bug-finding method in the last decade. In this process, developers and security practitioners fuzz a target program, identify crashes, triage these incidents, and patch the associated vulnerabilities. Typically, fuzzing and identifying crashes are automated, while triaging and patching are manual. After each patch, a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690278

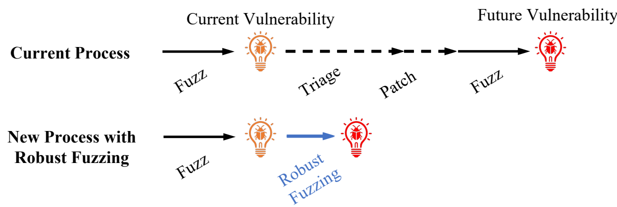

Figure 1: The pipeline of vulnerability discovery for the current process and that with robust fuzzing. The first row shows the current process, and the second row shows the discovery with robust fuzzing. Robust fuzzing speeds up the discovery of a future occluded vulnerability by skipping the wait of the occluding vulnerability's manual triage and patching.

new software version is released (or git commit committed), initiating another cycle of discovery, triage, and patching.

While this cycle works well in rapidly uncovering large numbers of vulnerabilities, fuzzing practitioners have noted the existence of "Fuzz Blocker" crashes: crashes that impede fuzzer's ability to execute code being *occluded* (like a cloud) by the bug [10]. Since fuzzing is a dynamic analysis and must trigger bugs to reason about them, Fuzz Blockers *serialize* the vulnerability discovery process — until these crashes are fixed, vulnerabilities occluded by them cannot be found. Further, the patch for the underlying bug must be implemented by expert human practitioners, rendering this cycle inefficient.

As an example, consider the infamous Stagefright vulnerabilities, a family of critical vulnerabilities that impacted an estimated 950 million Android devices running Android version 2.2 ("Froyo") through 5.11 ("Lollipop") and allowed attackers to perform arbitrary privileged execution simply by sending an MMS message with no end-user action [7]. This vulnerability remained undetected for four years after its introduction and was deeply embedded "in the heart of Android" [15]. It was not until all the superficial crashes were resolved that the core Stagefright bug was uncovered, resulting in more than ten CVEs and over twenty iterative and incomplete fixes, requiring three weeks of effort from a team of professional analysts [14].

In this paper, we *fuzz into the future*. Our goal is to proactively identify occluded future vulnerabilities, even when they are difficult to reach or completely occluded by latent bugs (e.g., `binutils` Issue 17531 [2] and Issue 20439 [3], which we will discuss in detail in Section 2). We do this by automating patching to save time spent waiting for occluding bugs to be fixed before occluded bugs can be found and thereby enhancing the overall effectiveness and speed of vulnerability discovery, as demonstrated in Figure 1.

Automatically patching bugs is difficult: the correctness of software patches depends on the semantics of the surrounding code and the specification of the program as a whole: e.g., a buffer overflow might be patched by increasing the buffer size, reducing the number of bytes accepted as input, fixing the size calculation, changing the buffer offset, or any number of ways depending on the specifics of the code. As a result, automated patching tools struggle to find the "correct" approach. However, previous work has suggested that even *semantically invalid* modifications to software can yield useful vulnerability analysis results [38].

Inspired by this, we realized that an entirely accurate patch for a crash is *not* always necessary: since many occluded future vulnerabilities do not depend on those previously discovered, a minimal patch that merely prevents the occluding crash from blocking further execution can be sufficient. Based on this insight, we developed a novel, vulnerability-tailored technique, *robust fuzzing*, which can be implemented in any fuzzing tool. Robust fuzzing allows fuzzers probe beyond the immediate crash location for common types of crashing bugs to uncover new vulnerabilities or variants of known ones, thus providing a more comprehensive understanding of the underlying issues.

To demonstrate robust fuzzing, we implemented it into the FlakJack prototype, built on AFL++. FlakJack is a dynamic binary patching technique that analyzes a crash and synthesizes a minimal patch to prevent the crash in future fuzzing attempts, and does so entirely without human assistance. FlakJack effectively synthesizes versions of a program that approximate future versions with valid fixes for any crashes discovered. By fuzzing the resulting binary, occluded future vulnerabilities can be found that would previously have been discovered only after the manual implementation of an accurate fix for the occluding crash.

Automatically repeating the fuzz–crash–patch cycle several times leads to discovering vulnerabilities much faster than traditional manual patching. As a result, multiple vulnerabilities can be fixed simultaneously, dramatically reducing the amount of time and effort required to test and release security fixes.

In our evaluation, we measured the effectiveness of FlakJack in expediting the discovery of occluded future vulnerabilities compared to a traditional fuzzer by testing FlakJack and AFL++ on old versions of 6 programs and comparing their discovery of occluded future vulnerabilities. FlakJack found 92 occluded future vulnerabilities while AFL++ found 24, an improvement of 3.8x that represents a cumulative vulnerability lifetime reduction (e.g., the sum of the release gap between the version we tested and the version in which the bug was fixed in real life) of over 37 *years*, at the cost of 3 false positive detections. We also applied FlakJack to up-to-date programs included in Google's OSS-Fuzz fuzzing suite [41], which helpfully marks certain unfixed bugs as Fuzz Blockers if it detects that they are being frequently triggered by the fuzzer. Starting from these blockers, FlakJack discovered 28 previously-unknown occluded vulnerabilities, as confirmed by a final manual triage. This shows that FlakJack is able to effectively find occluded future vulnerabilities that had gone undetected in projects that have been extensively tested by an industrial-scale fuzzer.

Contributions. In summary, we make the following contributions:

1. We propose a novel fuzz-into-the-future approach that can be applied to any fuzzer, called *robust fuzzing*, that uses dynamic patching to enable fuzzers to bypass crashes and discover occluded future vulnerabilities beyond the crash location.
2. We implement robust fuzzing, enhancing AFL++ into a prototype called FlakJack, a fuzzer particularly suited for occluded future vulnerability discovery.

```
1  static int display_debug_lines_raw (...){
2    ...
3    if (op_code >= linfo.li_opcode_base)
4    {
5      op_code -= linfo.li_opcode_base;
6      // ID 17531, DIVIDE-BY-ZERO
7      uladv = (op_code / linfo.li_line_range);
8      state_machine_regs.address += uladv;
9          }
10         else
11   ...
12 }
```

Listing 1: Binutils Vulnerability ID 17531 in function `display_lines_raw()`. A divide-by-zero vulnerability occurs in line 7.

(3) We demonstrate FlakJack's capability to discover occluded future vulnerabilities present in multiple real-world programs, finding over 37 years of bugs in old versions of software and 28 previously-unknown "future" vulnerabilities in up-to-date programs.

To foster open science, we have released the source code of the prototype at https://github.com/sefcom/flakjack.

2 MOTIVATION AND BACKGROUND

In this section, we introduce the concept of *occluded future vulnerabilities*. We will start with a motivating example, followed by the definition and related terms. We will then present the prevalence of occluded future vulnerabilities, and finally we will present the other related work and discuss the relationship between them and the fuzzing-the-future problem.

2.1 Motivating Example

As a motivating example, we present two binutils vulnerabilities: ID 17531 and ID 20439. Vulnerability ID 17531 is a divide-by-zero vulnerability, where variable `linfo.li_line_range` can be set to 0 at line 7 in function `display_debug_lines_raw`, as shown in Listing 1. This vulnerability will be triggered when the debug information in the binary is corrupted, resulting in a partial `.debug_line.` section being encountered without a prior full `.debug.line` section [1].

Vulnerability ID 20439 contains an overflow vulnerability occurring at line 8 in function `display_debug_lines_decoded` as shown in Listing 2. When a malformed debug information is fed into the program, variable `state_machine_regs.file` will be set to an invalid value. When the variable is used as a part of index expression for array `file_table` at line 8, a deference error will be triggered and thus the program crashes with segmentation fault.

Vulnerability 17531 occludes Vulnerability 20439 because 17531's vulnerable function `display_debug_lines_raw()` is executed before 20439's `display_debug_lines_decoded()`. As shown in Listing 3, both functions are under function `display_debug_lines()`. Therefore, in order to trigger vulnerability 20439, the condition at line 12 must *not* be satisfied, otherwise vulnerability 17531 will be triggered and the program will stop. Thus, there is less chance that a fuzzer can discover Vulnerability 20439.

```
1  static int display_debug_lines_decoded (...){
2    if ((is_special_opcode) ||
3        (op_code == DW_LNE_end_sequence) ||
4        (op_code == DW_LNS_copy))
5    {
6      const unsigned int MAX_FILENAME_LENGTH = 35;
7      // ID 20439, OVERFLOW
8      char *fileName = file_table[state_machine_regs.file - 1].name;
9      char *newFileName = NULL;
10     size_t fileNameLength = strlen (fileName);
11     ...
12   }
13 }
```

Listing 2: Binutils Vulnerability ID 20439 in function `display_debug_lines_decoded()`. An invalid index computed from malformed debug section information causes a segmentation fault at line 8.

```
1  static int
2  display_debug_lines (struct dwarf_section *section,
3                       void *file ATTRIBUTE_UNUSED) {
4    unsigned char *data = section->start;
5    unsigned char *end = data + section->size;
6    int retValRaw = 1;
7    int retValDecoded = 1;
8
9    if (do_debug_lines == 0)
10     do_debug_lines |= FLAG_DEBUG_LINES_RAW;
11
12   if (do_debug_lines & FLAG_DEBUG_LINES_RAW)
13     // The function containing Vulnerability 17531
14     retValRaw = display_debug_lines_raw (section, data, end);
15
16   if (do_debug_lines & FLAG_DEBUG_LINES_DECODED)
17     // The function containing Vulnerability 20439
18     retValDecoded = display_debug_lines_decoded (section, data, end);
19
20   if (retValRaw || retValDecoded)
21     return 0;
22
23   return 1;
24 }
```

Listing 3: Function `display_debug_lines()` in Binutils. Function `display_debug_lines_decoded()` can be executed after function `display_debug_lines_raw()`.

In reality, Vulnerability 20439 was not discovered until binutils 2.28 was released, which was 2 years after Vulnerability 17531 was discovered. However, we found that Vulnerability 20439 has already existed since binutils 2.21, which is 6 years before the discovery. As we applied fuzzing on binutils 2.21, we observed that the state-of-the-art fuzzers such as AFL++ were unable to detect vulnerability 20439. Our paper's goal is to find vulnerabilities like Vulnerability 20439, the *occluded future vulnerabilities* partially or completely occluded by another vulnerability in a target program.

2.2 Terms and Definition

As illustrated by the motivation example, in this paper, we define *occluded future vulnerabilities* as a class of vulnerabilities that are more likely to be found in the future than the present, because of another vulnerability/crash taking place on some or all paths that trigger the vulnerability. More formally,

Definition 2.1 (Occluded future vulnerabilities). A vulnerability V_2 at line L_2 is an occluded future vulnerability if there exists a vulnerability V_1 at line L_1 such that L_1 is executed before L_2 and thus V_1 prevents V_2 from being discovered.

Depending on the context, we also call future vulnerabilities as *occluded vulnerabilities* and use the two terms interchangeably. We name the vulnerabilities completely or partially blocking the discovery of the occluded future vulnerabilities *occluding vulnerabilities.* Occluded vulnerabilities have control flow dependency with the occluding vulnerabilities — either the occluding vulnerability dominates the basic block associated with the occluded vulnerabilities (full occlusion), or the occluding vulnerability exists in an earlier preceding basic block on some of the paths that trigger the occluded vulnerability (partial occlusion).

Note that occluded future vulnerabilities are *not* non-existent vulnerabilities, e.g., an indexed buffer element `a[index]` that might be overflowed only if the value of `index` is changed through future code. In our definition, occluded future vulnerabilities already exist in the program.

2.3 Prevalence of Occluded Future Vulnerabilities

A critical thinker might question the prevalence of occluded future vulnerabilities: are there indeed so many that we should be concerned? To answer this question, we conducted an experiment to estimate the occurrence of crash occlusion in `binutils` 2.21.[1] Note that these statistics are just for the purpose of demonstration and cannot precisely measure the number of occluded future vulnerabilities. A precise measurement would require addressing several research challenges, which we consider as separate and future work.

We first collected crashes for the most popularly fuzzed `binutils` targets using AFL++. We then deduplicated crashes using a hash computed from the backtrace at the time of crash. For every unique crash, we checked if fixing this crash will result in finding another different vulnerability. Specifically, we searched for the crash fixing commit; if found, we ran the crashing input with the patched program and checked if another crash is triggered and if the new crash is associated with a different vulnerability. If the check is passed, we consider the original crash as an occluding crash, i.e., a crash that will reveal another vulnerability if fixed.

Based on the measurement, we identified 345 unique crashes, among which 93 crashes are occluding crashes (Figure 2). This number indicates that, more than 25% of the identified crashes are occluding at least one occluded future vulnerability, not to mention the cases such as StageFright, which was occluded by 30+ crashes sequentially. These statistics imply that occluded future vulnerabilities can be quite prevalent in practice.

2.4 Related Work

Fuzzing for Vulnerability Discovery. Fuzzing [6, 8, 9, 16, 22, 23, 34, 40, 45] is a dynamic vulnerability discovery approach widely

[1]We selected an older version to ensure a sufficient number of total crashes, thereby making the results statistically representative.

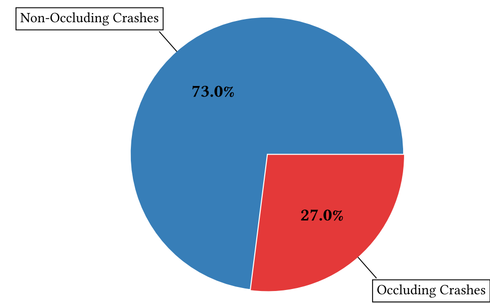

Figure 2: Occluding Crash Statistics in Binutils 2.21

used in real-world programs. The goal of fuzzing is to generate concrete inputs that trigger (enforceable) security violations through program execution.

Although both FlakJack and existing fuzzers are to find vulnerabilities, they share different focuses on the discovered vulnerabilities. While traditional fuzzers focus on finding current vulnerabilities, FlakJack aims to find occluded future vulnerabilities that are hard or impossible to trigger due to the existence of current vulnerabilities. FlakJack is built upon existing fuzzers, and the two fuzzing variants complement each other.

Additionally, AFL's crash exploration and Evocatio [27] are orthogonal to our work, as they aim to find more execution variants that trigger the same vulnerability, whereas FlakJack aims to discover different vulnerabilities. However, they can complement FlakJack by helping identify more cases to patch.

Vulnerability Prediction. Another stream of research that is possibly close to occluded future vulnerability discovery is vulnerability prediction [19, 28, 35, 46], which is to predict the introduction of vulnerabilities due to future code change. As we clarified in Section 2.2, occluded future vulnerabilities already exist in the program, which is different from the potential vulnerabilities considered in vulnerability prediction. Therefore, these two fields of research are orthogonal due to different research objectives.

Binary Patching and Crash Prevention. There has been a stream of research aimed at automatically patching binary programs [17, 31–33, 42, 47]. While the state of the art in binary patching shows promising advancements, existing techniques fall short of our goals due to due to incompatibility (e.g., needs as inputs vulnerability type [49] and/or location [20]), lack of generality (e.g., memory leaks only [43]), or high cost (e.g., requiring concolic execution [33, 48]). Moreover, automatic binary patching techniques are not sufficiently rapid to be directly applied to the "fuzz the future" problem, as generating a patch typically consumes excessive time and resources. Overall, current correctness-focused patching techniques overkill the robust fuzzing problem.

Existing crash prevention work such as X-Force [37] shares with robust fuzzing the goal of preventing program crashes, but it uses dynamic program instrumentation which significantly slows down program execution, which is critical for fuzzing that executes programs numerously. Therefore, there is a pressing need to develop

a new approach that achieves fast, automatic binary patching in robust fuzzing.

3 OVERVIEW

In this section, we provide an overview of the robust fuzzing technique. Figure 3 provides an overview of the overall system. The system starts after a fuzzer (which we call *base fuzzer* in this paper) discovers a crash. It analyzes the crash to determine the crash type and location as well as extracts information about the program state at the crash. Using this information, robust fuzzing generates a patch that prevents the crash from occurring again and determines an optimal location to insert this generated patch. It then switches to apply patch mode, inserting the earlier generated patch at the previously computed optimal location in the binary, preventing the crash from occurring again. If the patch is inserted successfully, the crashing input is added to the fuzzer's input queue and the technique switches to fuzzing mode with the newly generated patched binary as the target. If inserting the patch fails, it resumes fuzzing the previously used binary until another crash is discovered. This repeated Fuzz-Crash-Analyze-Patch approach enables robust fuzzing to continuously identify and patch vulnerabilities in the target binary, finding occluded future vulnerabilities.

Robust fuzzing is independent of which fuzzer or patching tool is used: it can work with any off-the-shelf fuzzer and binary patching tool. It also does not require source code, as all analyses are performed on the binary level.

3.1 Crash Verification and Triage

After the fuzzer has discovered a crash, the system verifies that the crash can reliably be reproduced. If the crash was reproduced successfully, it retrieves the memory mapping at the time of the crash. This memory mapping helps to identify the exact image where the crash occurs, which is required to identify the optimal location and type of patch to insert to prevent future occurrences of the crash. The memory mapping also helps identify static locations in the memory mapping; this information is used by certain types of patches inserted into the binary. The system also analyzes the crash to determine if it is of a type whose patching is currently supported. If either reproducing the crash fails, retrieving the memory mapping fails, or if the crash is of a type that cannot be patched, the system will not perform any further analysis and switch back to fuzzing mode until the next crash is discovered.

3.2 Patch Generation

If all the above steps succeed, the system proceeds with patch generation. Every patch generated by robust fuzzing has 3 key components:

(1) **Entry**: This is the entry point of the patch. Its objective is to analyze the current program state and determine if a crash is about to occur. The entry component also saves any program state that it overwrites for restoring later.
(2) **Crash preventer**: If the entry component determines that a crash will occur, control transfers to the crash preventer component. This component modifies the program state to prevent the crash.
(3) **Exit**: This is the final component of the patch that restores any changes made to the program state using the data saved by the entry component while preserving any changes made by the crash preventer.

In the following section, we will describe the three components of patches generated for all types of crashes currently supported by robust fuzzing.

4 PATCH GENERATION FOR ROBUST FUZZING

When designing the patching strategy, we considered two possible approaches - using a single, generic patch that can handle multiple crash types or a specific patch for each possible crash type. A significant advantage of a generic patch is that it can be reused several times without modification once designed. However, a generic patch has multiple limitations. Firstly, designing a generic patch that can handle multiple crash types can be particularly challenging given the diverse nature of possible crashes. Handling multiple crash types also increases the size of the generic patch, which could increase the difficulty of inserting the patch into a binary due to space constraints and introduce significant execution overhead. Additionally, adding support for a new crash type in the future could require significant modifications to the patch. On the other hand, specific patches can be tailor-made for particular crash types and thus can be relatively small. Designing patches for specific crash types simplifies adding support for more crash types in the future. However, this ease and flexibility come at the cost of an increased effort to design patches for every possible crash type.

In robust fuzzing, we adopted the latter approach of designing specific patches for each crash type. One of the main objectives of robust fuzzing is to help fuzzers discover valid occluded bugs that lie beyond a crash. This objective requires that the patches activate only when a crash is about to occur and perform the smallest modification needed to prevent the crash while preserving the rest of the program state. Using specific patches tremendously simplifies achieving this objective and, thus, an acceptable one-time cost.

The current design of robust fuzzing supports the following crashes: divide-by-zero, segmentation fault during memory access in the program under test (both memory read and memory write), segmentation fault at a function return, and crashes of two specific types in library functions.

Any other types of crashes are ignored and treated as patching failures. We restrict patching to only the target binary: no patches are inserted into shared code, such as library functions. Localizing patches to the vicinity of the crash and tailoring them to the specific crash enables us to precisely detect if a crash will occur and minimize the modifications made to the program state. Table 1 lists the supported types of crashes, a summary of the patch for each crash type, and the location where the patch is inserted. While robust fuzzing does not support all crash types, the supported types represent a large population of all crash types. Also, due to the design of the patch template, the robust fuzzing system can be easily extended for more crash types. In the rest of this section, we describe the design of patches for each crash type supported by our system.

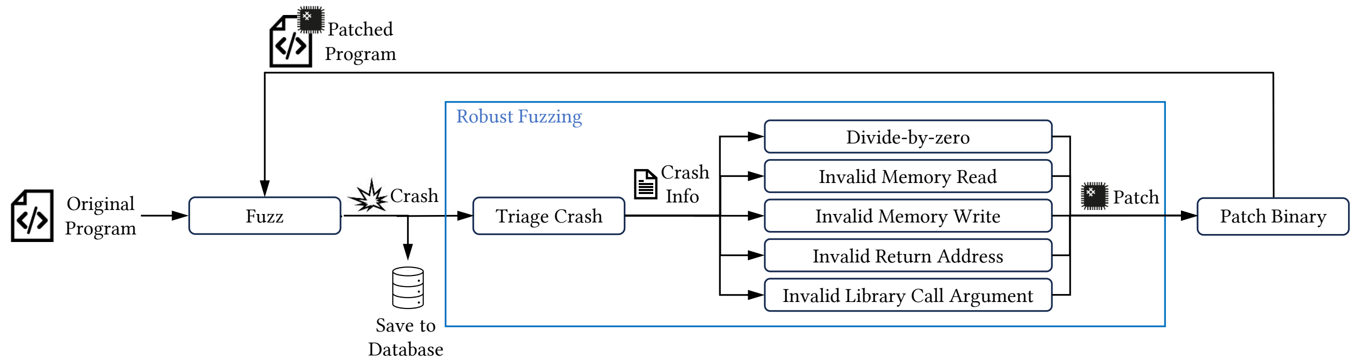

Figure 3: Robust fuzzing system overview

Table 1: List of supported crash types, summary of the patch for the crash and location of the patch

Signal Type	Crash Type	Patch Summary	Patch Location(s)
SIGFPE	Divide-by-zero in program	Set divisor to non-zero value	Before divide instruction
SIGSEGV	Memory read in program	Set address to a readable address	Before memory read instruction
SIGSEGV	Memory write in program	Set address to a writeable address	Before memory write instruction
SIGSEGV	Function return in program	Save and restore return address	Before entry and exit from function
SIGSEGV	Library function with 1 pointer argument	Set pointer to a readable address	Before function call
SIGSEGV	Library function with 2 pointer arguments	Skip function call	Before function call

```
1 cmp r15b, 0
2 jne nopatch
3 mov r15b, 55
4 nopatch:
```

Listing 4: Example patch for divide-by-zero

4.1 Divide-by-Zero

The idea of patching divide-by-zero is relatively simple: if the value of the divisor is 0, set the divisor to an arbitrary non-zero value. Listing 4 shows an example patch generated for a division operation performed with `r15b` register as divisor. The first two instructions belong to the entry component: they check whether the divisor `r15b` is 0. If yes, the crash preventer component sets `r15b` to a randomly generated non-zero value and ensures that a divide-by-zero does not occur. Since the check performed by the entry component is straightforward, no program state cleanup is required, and thus, the component is not present in this patch. The robust fuzzing system supports patching divide-by-zero exceptions arising from register and memory operand divisors.

4.2 Segmentation Fault at Memory Read

A segmentation fault occurs at a memory read because the address being read from is invalid. Listing 5 shows the patch generated for a segmentation fault at the instruction `movzx eax, word [rax+rcx*2]`. The patch performs the following operations:

(1) The entry component saves any registers that will be clobbered for use by the exit component (lines 1 to 6).
(2) Next, the entry component opens a file on disk in write-only mode (lines 11 to 18), writes two bytes from the address in question (line 8) to the open file (lines 22 to 25), and closes the file.
(3) Next, it checks if two bytes were successfully written to the file (lines 33 to 34). If yes, the exit component is executed since the memory read will not trigger a segmentation fault; thus, no patching is required.
(4) If the two bytes were not written successfully, patching is required as a memory dereference will trigger a segmentation fault. The crash preventer sets the base register(here `rax`) to a valid address, from where two bytes can be read, and all other registers(here `rcx`), if any, to zero. This address is randomly chosen from a static readable location in the memory mapping extracted from the program at the time of the crash and is guaranteed to be a valid address during any point in the program execution.
(5) The exit component is executed (lines 38 to 44 or lines 47 to 52, depending on whether patching was required) to restore any program state modified by the entry component. A key thing to note in this example is that the `rax` register is restored by the exit component only if the crash preventer did not execute (line 38 vs line 47).

4.3 Segmentation Fault at Memory Write

Unlike memory reads, a segmentation fault at memory write can occur for two reasons: the address being written to is either invalid or not writeable. Thus, we cannot use the patch used for segmentation faults at memory reads to handle segmentation faults at memory

```
push rcx    ; clobbered by syscall instruction
push r11    ; clobbered by syscall instruction
push rdx
push rsi
push rdi
push rax
; save mem address value on stack
lea rsi, [rax + rcx * 2]
push rsi
; open("/tmp/fj", O_CREAT | O_WRONLY)
mov rsi, 0x6a662f706d742f
push rsi
xor edx, edx
mov esi, 65
mov rdi, rsp
mov eax, 2
add rsp, 8
syscall
pop rsi
push rax
; write(<fd>, <addr>, <size>)
mov rdx, 2
mov rdi, rax
mov eax, 1
syscall
; close(<fd>)
pop rdi
push rax
mov eax, 3
syscall
; if `size` bytes written, do not patch
pop rax
cmp eax, 2
je nopatch
; patch component
  mov rax, 0x3eca7b
  mov rcx, 0
  add rsp, 8
  pop rdi
  pop rsi
  pop rdx
  pop r11
  add rsp, 8
  jmp done

nopatch:
  pop rax
  pop rdi
  pop rsi
  pop rdx
  pop r11
  pop rcx
done:
```

Listing 5: Example patch for segmentation fault at memory read in the program

write. We slightly modified the patch for a memory read to account for both the possibilities for a memory write as explained below. Listing 6 shows the patch generated for a segmentation fault at the instruction `mov dword [rdx+rcx*4],eax`. The patch performs the following operations:

(1) Similar to the memory read patch, the entry component saves any registers clobbered for restoring later (lines 1 to 6).
(2) Next, `/dev/random` is opened in read-only mode (lines 11 to 20), 4 bytes are read from `/dev/random` into the memory location with address `rdx + rcx*4` and the file is closed.
(3) If the four bytes were not read successfully, then the address `rdx + rcx * 4` is not writeable and patching is required. The crash preventer sets the base register(here `rdx`) to a valid writeable address, to which 4 bytes can be written, and all other registers(here `rcx`), if any, to zero. Similar to the patch for segmentation fault at a memory read, the address is chosen randomly from a static writeable location in the memory mapping extracted from the program at the time of the crash and thus guaranteed to be a valid address at any point in the program execution.
(4) Finally, the exit component is executed (lines 40 to 46 or 48 to 53, depending on if patching was required) to restore any program state. As in the previous example, `rdx` and `rcx` are left unmodified if the crash preventer component was executed.

4.4 Segmentation Fault at Function Return

A segmentation fault occurs when returning from a function where the saved return address was overwritten with an invalid address. Unlike other cases, we use two patches to handle this crash (Listing 7 and 8). The first patch is inserted at the start of the function, which saves the return address to a fixed location(here address `0x7000000`). The second patch is inserted just before the `ret` instruction, which checks if the current return address matches the original return address. If they do not match, the original return address is restored, and execution resumes.

4.5 Segmentation Fault in Libraries

Segmentation faults in library functions are typically caused by the arguments passed to them. The robust fuzzing system supports patching crashes at two types of functions: those that take a single argument of pointer type (e.g., `strlen`, `strchr`, etc) and those that take two arguments of pointer type as arguments(e.g., `memcpy`, `memmove`, `memcmp`, etc).

Consider the patch in Listing 9 for a crash in `strlen`.

(1) Similar to in the memory read patch, the entry component saves any registers clobbered for restoring later (lines 1 to 6).
(2) Next, the component uses the Linux `madvise` system call to determine if the memory page to which the pointer belongs is a valid, mapped memory page (lines 8 to 16). If the page is not mapped, then patching is required.
(3) The crash preventer modifies `rdi` (which contains the first argument to the function as per the calling convention) to point to a valid mapped page.
(4) Finally, the exit component performs the necessary cleanup (lines 22 to 28 or 23 to 28, depending on whether the crash preventer was executed).

While, in theory, the sole pointer argument could span more than one page, from our experiments, we found that checking for a single page was sufficient.

Listing 10 shows a patch for a crash in `memcpy` invoked at address `0x258f39`.

```
push rcx    ; clobbered by syscall instruction
push r11    ; clobbered by syscall instruction
push rdx
push rsi
push rdi
push rax
; save mem address value on stack
lea rsi, [rdx + rcx * 4]
push rsi
; open("/dev/random", O_RDONLY)
mov esi, 0x6d6f64
push rsi
mov rsi, 0x6e61722f7665642f
push rsi
xor edx, edx
mov esi, 0
mov rdi, rsp
mov eax, 2
add rsp, 16
syscall
pop rsi
push rax
; read(<fd>, <addr>, <size>)
mov rdx, 4
mov rdi, rax
mov eax, 0
syscall
; close(<fd>)
pop rdi
push rax
mov eax, 3
syscall
; if `size` bytes read, do not patch
pop rax
cmp eax, 4
je nopatch
; patch component
  mov rdx, 0x42d191
  mov rcx, 0
  pop rax
  pop rdi
  pop rsi
  add rsp, 8
  pop r11
  add rsp, 8
  jmp done
nopatch:
  pop rax
  pop rdi
  pop rsi
  pop rdx
  pop r11
  pop rcx
done:
```

Listing 6: Example patch for segmentation fault at memory write in the program

```
push rdi
mov rdi, [rsp + 8]
mov [0x7000000], rdi
pop rdi
```

Listing 7: Example patch for segmentation fault at ret instruction in the target binary (function entry)

(1) After saving any clobbered registers (lines 1 to 9), the entry component checks if all bytes of the source and the destination belong to a mapped page in memory using the `madvise` syscall (lines 16 to 27 and 31 to 43).

```
push rsi
push rdi
mov rdi, [rsp + 16]
mov rsi, [0x7000000]
cmp rdi, rsi
je nopatch
mov [rsp + 16], rsi
nopatch:
  pop rdi
  pop rsi
```

Listing 8: Example patch for segmentation fault at ret instruction in the target binary (function exit)

```
push rcx    ; clobbered by syscall instruction
push r11    ; clobbered by syscall instruction
push rdx
push rsi
push rax
push rdi
; madvise(<address of page with string>, <page_size>, MADV_NORMAL)
mov rdx, 4095
not rdx
and rdi, rdx
mov rsi, 4096
mov rdx, 0
mov rax, 28
syscall
cmp eax, 0
jge nopatch
; patch component
  mov rdi, 2379059
  add rsp, 8
  jmp done
nopatch:
  pop rdi
done:
  pop rax
  pop rsi
  pop rdx
  pop r11
  pop rcx
```

Listing 9: Example patch for segmentation fault at strlen

(2) If the source or the destination have any bytes that do not belong to a mapped page, the crash preventer skips the `memcpy` call and continues from the next instruction after the call (at address `0x258f3e`).

(3) It should be noted that the crash preventer component executes after the exit component performs its cleanup since the fix by the crash executor to prevent the crash is to skip invoking `memcpy` altogether.

5 IMPLEMENTATION

We implemented the robust fuzzing system and developed FlakJack, a fuzzing add-on designed to discover occluded future vulnerabilities. We utilized AFL++ v4.09c [18] as the foundational fuzzer and Patcherex [5] for binary rewriting. Additionally, we employed GDB interfaces to gather crash information, which aids in triage and generates crash type-specific patches. FlakJack generates binary-only patches, supports binary-only targets, and is compiler-agnostic. The current implementation focuses on x86_64 binary programs.

```
1  push rcx    ; clobbered by syscall instruction
2  push r11    ; clobbered by syscall instruction
3  push rdx
4  push rsi
5  push rax
6  push rdi
7  push rbx
8  push r12
9  push r13
10 ; save arguments for use later
11 mov rbx, rdi
12 mov r12, rsi
13 mov r13, rdx
14 ; check if all destination bytes are valid using madvise
15 ; compute start address of page with first byte of destination pointer
16 mov rdx, 4095
17 not rdx
18 and rdi, rdx
19 ; compute end address of page with last byte of destination pointer
20 ; and thus, number of bytes to check
21 lea rsi, [rbx + r13 + 4096]
22 and rsi, rdx
23 sub rsi, rdi
24 mov rdx, 0
25 mov rax, 28
26 syscall
27 cmp eax, 0
28 jl skip_memcpy
29 ; check if all source bytes are valid using madvise
30 ; compute start address of page with first byte of source pointer
31 mov rdi, r12
32 mov rdx, 4095
33 not rdx
34 and rdi, rdx
35 ; compute end address of page with last byte of source pointer and thus,
36 ; number of bytes to check
37 lea rsi, [r12 + r13 + 4096]
38 and rsi, rdx
39 sub rsi, rdi
40 mov rdx, 0
41 mov rax, 28
42 syscall
43 cmp eax, 0
44 jge done
45 skip_memcpy:
46     pop r13
47     pop r12
48     pop rbx
49     pop rdi
50     pop rax
51     pop rsi
52     pop rdx
53     pop r11
54     pop rcx
55     jmp 0x258f3e
56 done:
57     pop r13
58     pop r12
59     pop rbx
60     pop rdi
61     pop rax
62     pop rsi
63     pop rdx
64     pop r11
65     pop rcx
```

Listing 10: Example patch for segmentation fault at memcpy

FlakJack is an extremely lightweight tool, comprising approximately 1200 lines of Python code. It interfaces with AFL++ through the Phuzzer library [4] and GDB via the GDB Python API provided by pwntools [21].

6 EVALUATION

To evaluate the effectiveness of robust fuzzing in uncovering occluded future vulnerabilities, we designed two experiments for FlakJack to address the following research questions:

(1) How effective is robust fuzzing at accelerating the discovery of *occluded future vulnerabilities* when integrated with an existing fuzzing engine (Section 6.1)?
(2) Is robust fuzzing capable of detecting future bugs in current real-world projects (Section 6.2)?

Both experiments adhere to established fuzzing norms. Specifically, we conducted all experiments on Ubuntu 22.04 LTS with a memory cap of 8GB. In line with recommendations from the fuzzing evaluation research by Klees et al. [29], each experiment was run for 24 hours and repeated ten times.

All target programs were compiled with static linkage for all project code and dynamic linkage for all external libraries (e.g., `libc`), following the guidelines suggested by the developers of AFL++ for fuzzing applications using AFL++ [12].

6.1 FlakJack's Acceleration in Occluded Future Vulnerability Discovery

We conducted an experiment to assess the effectiveness of robust fuzzing in accelerating the discovery of occluded future vulnerabilities when integrated with an existing fuzzing engine. At a high level, this experiment involved running FlakJack, the implementation of robust fuzzing, alongside its base fuzzer, AFL++ [18], on a set of target programs from real-world projects. We followed the fuzzing process mentioned above, counted the number of occluded future vulnerabilities identified by FlakJack and by AFL++, and compared the results generated by the two tools.

Dataset. As our paper pioneers the investigation of occluded future vulnerabilities, there is no established dataset for occluded future vulnerabilities currently. Therefore, we need to construct a comprehensive dataset to effectively evaluate robust fuzzing and potentially future techniques.

One possible approach would be to collect a random corpus of current projects. However, this method may not be suitable because current fuzzing techniques are not effective enough for all programs. For instance, programs that extensively use encryption or those for which crafting high-quality harnesses is challenging may not yield a sufficient number of occluded future vulnerabilities for meaningful statistical comparison when tested with the base fuzzer and FlakJack. Therefore, we need to select projects known to have vulnerabilities previously identified by our comparison base fuzzer, AFL++.

Additionally, if a project has undergone intensive fuzzing, choosing its most recent version might introduce bias since vulnerabilities detectable by the base fuzzer may have already been patched, making it difficult to find further vulnerabilities. To avoid bias from projects that have been extensively fuzzed, we need to find programs that are known to be amenable to fuzzing but have not been heavily subjected to it by the base fuzzer or similar tools.

Considering these factors, we decided to utilize a set of earlier versions of fuzzing targets known to contain a significant number of vulnerabilities. In this way, we will obtain historical fuzzer-found

vulnerabilities, and treat as future vulnerabilities the later vulnerabilities that have been found in the version that patches those earlier vulnerabilities. Specifically, we built the dataset through the following steps:

(1) We surveyed fuzzing papers published in the top four security conferences since 2020 to identify the programs used in their evaluation.
(2) From this list, we chose programs that have at least 100 CVEs reported from 2012 as per the National Vulnerability Database [36] and built the oldest successfully buildable version of each project using AFL++ in LTO mode [26], which is the recommended source instrumentation mode by AFL++ developers.
(3) For projects successfully built in the previous step, we chose the most commonly used programs from these projects in previous papers where AFL++ can find at least one bug of a type that is patchable by FlakJack.
(4) For each program, we selected the command line arguments and seeds from the Unifuzz dataset [30]. For programs not present in the Unifuzz dataset, we pick valid seeds from the dataset and the most commonly used command line arguments from previous security bugs reported to the project.

This process ensures that we have the projects widely tested by fuzzers, with several security-critical bugs reported in the past and a sufficiently long development history that enables reliably measuring FlakJack's performance of finding occluded future vulnerabilities over years and versions. In this way, this experiment essentially investigates that, if FlakJack were added to a base fuzzer when fuzzing a program at version X, can it identify occluded future vulnerabilities more effectively than the base fuzzer. We list the projects and specific programs chosen using the above method in Table 2.

Table 2: The testing projects and their version selected throughout the process.

Binary Tested	Project	Version	Release Year
nm	binutils	2.21	2009
objdump			
readelf			
ffmpeg	ffmpeg	0.10.1	2012
MP4Box	gpac	0.7.0	2017
tiffcp	libtiff	4.0.1	2012

Occluded Future Vulnerability Identification. Recall that an occluded future vulnerability is a vulnerability that is occluded (fully or partially) by another vulnerability. Following the definition, for each crash discovered by FlakJack in each binary, we determine if the crash is associated with a true occluded future vulnerability as follows:

(1) *False positive crash checking.* We automatically verify if the crashing input causes the original binary to crash. If not, we deem the crash as a false positive (§ 6.1) (which is caused by the design of approximate patching).
(2) *Occluded future vulnerability checking.* We automatically verify if the crashing input causes the original binary to stop at an earlier crashing point.
(3) *False positive occluded future vulnerability checking.* We verify if the identified occluded future vulnerability exists in the original binary. We employ a hybrid approach for the verification, where we first find the fixing commit for the crash in the original version and check if the crashing input triggers a crash in the fixed binary. If this fixed binary crashes and the crash is identical to the crash in the FlakJack generated patched binary, then crash triggered by FlakJack is an occluded bug. Otherwise, we manually check if the identified occluded future vulnerability exists if the original binary is fixed for the occluding crash. If not present, we will label the crash as a false positive.

Discovered Occluded Future Vulnerability Statistics. Table 3 displays the number of occluded future vulnerabilities identified by FlakJack and its base fuzzer, AFL++. Note that AFL++ still detected some occluded future vulnerabilities because these were not fully occluded. In total, among all 92 occluded future vulnerabilities identified by FlakJack, AFL++ only discovered 24 vulnerabilities, which is 26% the number discovered by FlakJack. These results clearly demonstrate that FlakJack significantly complements its base fuzzer in detecting more occluded future vulnerabilities. With the implementation of robust fuzzing, the effectiveness of discovering occluded future vulnerabilities has been markedly enhanced.

We further analysed all occluded bug to determine the occlusion depth i.e. how many occluding crashes need to be overcome before the occluded bug is reached. We found that on average, at least 2 unique crashes need to be bypassed in order to reach an occluded bug with a maximum of 20 crashes in some cases. Table 3 also lists the average, median and maximum number of occluding crashes to be bypassed before occluded bugs are triggered for each target.

Table 3: Number of occluded future vulnerabilities discovered by FlakJack and AFL++.

Target	FlakJack	AFL++	Occlusion depth		
			Mean	Median	Max
nm	7	5	1	1	2
objdump	25	1	2.25	2	20
readelf	31	12	2	2	17
ffmpeg	15	2	4.2	4	8
MP4Box	12	3	2.2	1	4
tiffcp	2	1	1.75	2	3
Total	92	24	1.86	1	20

FlakJack identified both new vulnerabilities and new variants of known ones. For example, the motivating example we introduced in Section 2 was exclusively found by FlakJack. Besides that, FlakJack also identified polymorphic vulnerabilities, such as the divide-by-zero vulnerability in function `display_debug_lines_raw`, Vulnerability ID 17531 [2] (Listing 1). In the meantime, there is an identical vulnerability in function `display_debug_lines_decoded`, as shown in Listing 11 at Line 8.

```
1  /* This loop iterates through the Dwarf Line Number Program.  */
2  while (data < end_of_sequence)
3  {
4    ...
5    if (op_code >= linfo.li_opcode_base)
6    {
7      ...
8      adv = (op_code % linfo.li_line_range) + linfo.li_line_base;
9      state_machine_regs.line += adv;
10     is_special_opcode = 1;
11   }
12   ...
13 }
```

Listing 11: Function `display_debug_lines_decoded` in binutils. A SIGFPE occurs at line 8 due to a divide-by-zero vulnerability

```
1 lea    rdx,[rdi+rcx*4]
2 mov    QWORD PTR [rsp-0x30],rdx
3 ...
4 mov    rdi,QWORD PTR [rsp-0x30]
5 movzx  eax,BYTE PTR [rdi+rbx*1]
```

Listing 12: Assembly code for `cavs_idct8_add_c` in ffmpeg

We observed that AFL++ failed to trigger the crash in function `display_debug_lines_decoded` because this function is occluded by `display_debug_lines_raw` in nearly all potential executions, as illustrated in Listing 3. In contrast, FlakJack successfully identified both crash variants. After encountering the first variant, FlakJack applied a patch and continued operation, executing an additional 17,744 basic blocks before encountering the second crash. These executions spanned across 44 unique basic blocks, demonstrating that FlakJack is capable of identifying vulnerabilities or their variants that are significantly distant from each other in the code.

False Positives. Unlike traditional fuzzing, robust fuzzing can lead to false positives — instances where inputs cause the patched binary to crash but not the original binary — due to approximations in the patching process. We quantified the number of false positives produced by FlakJack, and we found that FlakJack generated three false positives from ffmpeg and none from the other five programs.

We further investigated the three false positives and discovered that they were all related to a single issue which arises when a patch is inserted into function `cavs_idct8_add_c`. A snippet of the assembly code from the function `cavs_idct8_add_c` is shown in Listing 12. In this snippet, some pointers are stored at addresses that are at a negative offset to the stack pointer. Later on in this function, these pointers are retrieved by using the same negative offset and are dereferenced.

When FlakJack inserts a patch into a function, it assumes that all local variables and pointers used by the function are located at non-negative offsets from the stack pointer. In this highly unusual scenario, however, this assumption is incorrect and the patch ends up overwriting the pointers below the bottom of the stack. When these pointers are eventually dereferenced, it leads to a segmentation fault due to the overwritten value, as shown Figure 4.

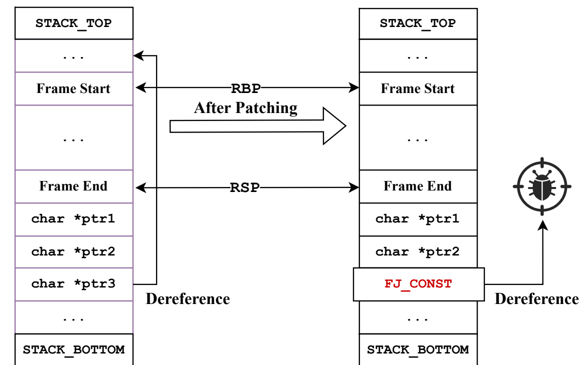

Figure 4: The false positive crash in ffmpeg that arises due to pointers being stored outside of the function's stack frame.

Patching Performance and Representativeness. Recall that a primary objective of FlakJack is to facilitate rapid patching. To assess the patching performance, we measured the average patching speed of FlakJack across each fuzzing process, and we compute the average and standard deviation for the average patching speed. Our results indicate that FlakJack consistently patched any target program within 100 seconds. Additionally, the variability in patching time across all fuzzing targets was marginal, with the standard deviations ranging from 0.00014 to 0.001. These findings suggest that FlakJack reliably achieves rapid patching across a variety of programs.

Furthemore, we assessed the patching sucess rate over all target programs. In this experiment, FlakJack generated patches for 7582 out of 8093 patching tasks, with a success rate over 93% . This number indicates that the patches that we design for robust fuzzing are representative for real-world targets.

Case Study: Occluded Future Vulnerability Discovery Acceleration from Historical Binutils Project.

We further explored the occluded future vulnerabilities exclusively identified by FlakJack and assessed the potential time savings in detecting these vulnerabilities compared to their historical discovery timeline. Specifically, we analyzed the lifetime of six occluded future vulnerabilities in the project `binutils`, exclusively identified by FlakJack. We sourced the report and corresponding fixes for these vulnerabilities from public records. The timeline was calculated from the release date of our target program — when FlakJack first identified the vulnerability — to the official report date.

Table 5 presents the details of the investigated vulnerabilities for `binutils` version 2.21, including the commit of the report, the reporting date, the earliest release of `binutils` incorporating the fix, and the duration from the release date of the `binutils` version analyzed by FlakJack to the actual report date of the vulnerability. According to the table, the occluded future vulnerabilities in `binutils` 2.21 identified by FlakJack were not recognized until the release of `binutils` 2.26 or later. These vulnerabilities, discovered between 2014 and 2016, could have been identified as early as 2009 had FlakJack been deployed, as illustrated in Figure 5. Overall, FlakJack has the potential to save approximately 35+ engineering years for merely these six vulnerabilities.

Case study: Analyzing exploitability of occluded bugs

To further understand the benefit of FlakJack in fuzzing, we analyze the FlakJack discovered occluded bugs to understand their exploitability. Specifically, we analyze the control flow graph (CFG) recovered from the compiled programs to determine if the occluded bugs are occluded along all possible valid execution paths within the program or if they can be triggered through an alternate execution path. Table 4 lists the count of partially and fully occluded bugs discovered by FlakJack.

Table 4: Statistics of partially and fully occluded vulnerabilities exclusively discovered by FlakJack.

Target	Fully occluded	Partially occluded
ffmpeg	12	3
mp4box	5	7
nm	2	5
objdump	23	2
readelf	19	12
tiffcp	1	1

From table 4, we see that most of the bugs are fully occluded i.e. there exist no valid execution paths through the program which can trigger the occluded bug. Thus, these bugs are currently not exploitable since they cannot be triggered but future changes to the program could fix the occluding bug, making the occluded bug triggerable and potentially exploitable. However, some of the undiscovered bugs are partially occluded i.e. they can be triggered via at least one execution path along which no vulnerabilities exist. Thus, the partially occluded bugs can likely be triggered in the unmodified binary and possibly exploitable. By fixing the occluding vulnerability, FlakJack enabled the fuzzer discover such partially occluded bug, which was otherwise undiscovered. Thus, FlakJack not only enables discovering fully occluded vulnerabilities that may be exploitable in future but also enables discovering currently exploitable vulnerabilities that are difficult for a fuzzer to trigger.

6.2 Finding Occluded Future Vulnerabilities from Current Real-world Projects

In this experiment, we operated FlakJack on real-world programs with the objective of uncovering occluded future vulnerabilities in the present. Unlike the projects selected in the previous experiment, here we chose projects from OSS-Fuzz due to the availability and accessibility of unfixed crashes. The OSS-Fuzz issue tracker features several unresolved issues labeled as "Fuzz-blocker," i.e., crashes that occur frequently during fuzzing and thus blocking the fuzzing process.

We analyzed all reproducible, fuzz-blocking crashes and selected those that could be patched by FlakJack. Considering that OSS-Fuzz harnesses utilize sanitizers (which influence crash triaging in FlakJack) and can be built for different fuzzers, we specifically filtered these crashes to include only those fuzz harnesses that could be built for AFL++ without sanitizers, resulting in 17 bugs across 15 projects.

For each fuzzer harness, we executed FlakJack using the same seeds and configuration as employed by OSS-Fuzz. Differently from the previous experiment, we initiated the process with the Fuzz-blocker crash from OSS-Fuzz as the first crashing input, patched this crash, and then continued the operation as planned. We employed the same methodology to identify occluded future vulnerabilities as detailed in the previous experiment, where we repeated the fuzzing process for 10 times, collected all the identified crashes, and screened them by the rules in order to obtain occluded future vulnerabilities.

```
1 Page *Catalog::getPage(int i) {
2   Page *page;
3   if (!pages[i-1]) {
4     loadPage(i);
5   }
6   page = pages[i-1];
7   return page;
8 }
```

Listing 13: Function `getPage` in xpdf OSS-Fuzz harness. An invalid page can be accessed if `i` is 0

Table 6 provides the details of the discovery, including the name of the target project, the OSS-Fuzz issue ID of the crash input for FlakJack to start, and the number of occluded future vulnerabilities FlakJack identified from the target project and the Fuzz-blocker crash. FlakJack successfully identified 28 occluded future vulnerabilities from 9 current projects in total. Notably, FlakJack discovered 14 and 7 occluded future vulnerabilities in `inchi` and `libavc`, respectively. These results affirm that FlakJack is capable of identifying occluded future vulnerabilities in current real-world programs.

Furthermore, the results underscore the critical importance of addressing Fuzz-blocker crashes in practice. The rich number of identified occluded future vulnerabilities illustrate the potential impact caused by fuzzing-blockers: a significant number of vulnerabilities can be uncovered if these blocks are resolved. Thus, Fuzz-blocker crashes do more than merely slow down or impede the fuzzing process — more importantly, they prevent the exploration of certain program areas, thereby degrading the overall effectiveness of vulnerability discovery.

False Positives. In this experiment, FlakJack generated 39 false positive bugs. While most of the bugs are caused by FlakJack's approximation on patches, we also identified an interesting case from project xpdf, as shown in Listing 13 and in project haproxy. In this case, variable `i` can be set to 0, and thus a crash happens because of invalid access `pages[-1]`. As we looked into this issue, we realized that the buggy program comes from the fuzzing harness `fuzz_pdfload.cc` instead of xpdf's executable program, which was just reported one month ago [11]. Although the identified crash is a false positive, FlakJack still helped identify a valid issue related to the project. In case of haproxy, we discovered that all 29 bugs discovered in haproxy are not valid bugs because several initial checks have been skipped in the fuzz harness. We also found that this issue in fuzzing harness was previously discovered by developers when analyzing a different bug also reported using the OSS-Fuzz fuzzing harness [24].

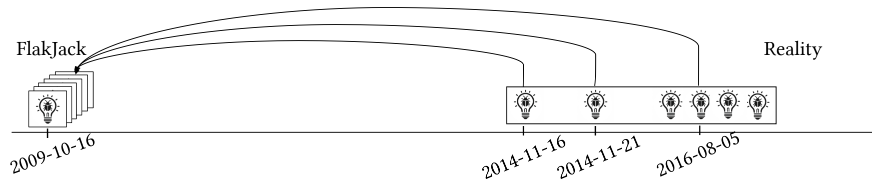

Figure 5: Occluded Future Vulnerability Discovery Timeline: The world with FlakJack and without (Reality). All the 6 different vulnerabilities discovered since year 2014 could have been discovered in 2009 by FlakJack.

Table 5: Details of occluded future vulnerabilities discovery by FlakJack and the reality.

ID	FlakJack		Reality			Time Saved by FlakJack
	Target Version	Target Release Date	Target Version	Fix Commit	Bug Report Date	
5375	binutils-2.21	2009-10-16	binutils-2.28	db9537d2b73	2016-08-05	6 years 9 months 20 days
30085			binutils-2.28	db9537d2b73	2016-08-05	6 years 9 months 20 days
66725			binutils-2.28	db9537d2b73	2016-08-05	6 years 9 months 20 days
179			binutils-2.28	db9537d2b73	2016-08-05	6 years 9 months 20 days
77529			binutils-2.26	6937bb54a9c	2014-11-16	5 years 1 months 0 days
48071			binutils-2.26	0a9d414aa11	2014-11-21	5 years 1 months 5 days

Table 6: Discovered Occluded Future Vulnerabilities from Current Projects

Project	OSS-Fuzz Issue ID	#Occluded Future Vulnerability
assimp	58667	0
cairo	54783	0
haproxy	36266	0
inchi	37224	14
libavc	55608	7
libbpf	62476	0
libsass	31594	0
libsass	47248	2
netcdf	38537	2
sleuthkit	50396	0
tidy-html5	36781	0
tremor	19619	0
tremor	19860	0
tremor	19872	0
unit	52469	1
wasm3	40923	2
xpdf	44461	0
Total		**28**

7 DISCUSSION AND FUTURE WORK

Building an Occluded Future Vulnerability Benchmark. In our current evaluation, we have not measured the false negatives, i.e., the number of occluded future vulnerabilities that FlakJack and its base fuzzer missed in target programs. This omission stems from the challenge of obtaining the ground truth about occluded future vulnerabilities in real-world projects, which is hindered by a lack of publicly available information, such as vulnerability-triggering inputs and fixing commit logs. Moreover, completely identifying all occluded future vulnerabilities is currently impossible, as no technique yet exists that can identify all execution paths that trigger a vulnerability.

Furthermore, constructing such a benchmark is challenging. A potential solution could involve building a dataset based on real-world programs. However, few real-world projects meet all necessary criteria, including 1) sufficient age to include a comprehensive set of occluded future vulnerabilities, 2) compatibility with environments that support modern fuzzers, and 3) presence of surface-level crashes that can be readily discovered by fuzzers. Thus, we propose the development of an occluded future vulnerability benchmark as a separate research project, akin to projects such as LAVA [13] and MAGMA [25].

Support for More Crash Types and Architectures. The current robust fuzzing system can be extended to support additional types of crashes. For instance, in our analysis of crashes in tiffcp, ffmpeg, and MP4Box, we identified several crashes related to floating point operations (e.g., XMM and AVX instructions), dynamic memory allocation functions (e.g., `malloc`, `free`, `new`), and indirect control transfer instructions (e.g., indirect calls and jumps). Unfortunately, the system does not currently support patching these types of crashes. Additionally, FlakJack only supports x64 and ELF executables. However, the core concept is adaptable to any platform or architecture, and we plan to explore these extensions in future work.

Issues Caused by Dependent Tools. In our implementation, FlakJack uses GDB to triage crashes and extract necessary information for generating patches. However, some crashes are not reproducible with GDB; we attribute this to limitations in reproducing the crash in GDB or to the remote procedure-call-based interface of pwntools' Python API to GDB. This issue might be mitigated by methods such as deterministic execution replaying or time-travel debugging (e.g., rr [39]).

FlakJack relies on a specific technique in Patcherex to apply generated patches to the binary. However, this technique occasionally fails to insert patches and also relocates some valid code around the crash site to a different section, affecting crash triaging. We believe that these limitations can be addressed by modifying the current technique to handle these problem cases, a task orthogonal to the core functionality of FlakJack. Alternatively, approaches such as Ramblr [44] could help to overcome these limitations.

Currently, FlakJack does not work well with sanitizers because their instrumentation impacts the required triaging. However, sanitizers are extensively used alongside fuzzers today and provide additional information about crashes. We believe that modifying FlakJack to work with sanitizers is an interesting idea that could improve the quality of patches generated.

8 CONCLUSION

In this paper, we introduce the concept of occluded future vulnerabilities, a class of vulnerabilities whose execution paths are either completely or partially occluded by other vulnerabilities. We highlight the significance of identifying such vulnerabilities and propose a method for their detection. Building on this method, we created robust fuzzing and developed FlakJack, which enhances the capability of its base fuzzer to accelerate the discovery of occluded future vulnerabilities. Using FlakJack, we successfully identified 28 new vulnerabilities in projects that are actively tested in real-world scenarios. The development of FlakJack represents a significant breakthrough in software security, offering more efficient and effective management of vulnerabilities.

ACKNOWLEDGMENTS

This material is based upon work supported by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-22-C-4026, the Department of Defense under Contract No. H98230-23-C-0270, the National Science Foundation under Contract Nos. 2146568, 2232915 and 2247954 and the Defense Advanced Research Projects Agency (DARPA) under Contract No. D22AP00145-00. Approved for public release; distribution is unlimited. The project or effort depicted was or is sponsored by the Defense Advanced Research Projects Agency, the content of the information does not necessarily reflect the position or the policy of the Governments, and no official endorsement should be inferred. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ARPA-H, DARPA or NIWC Pacific.

REFERENCES

[1] 2014. binutils #17531 fix. Commit ID 058037d3a16 changeset.
[2] 2014. binutils issue #17531. https://sourceware.org/bugzilla/show_bug.cgi?id=17531#c57.
[3] 2016. binutils issue #20439. https://sourceware.org/bugzilla/show_bug.cgi?id=20439.
[4] angr team. 2015. Phuzzer. https://github.com/angr/phuzzer/.
[5] angr team. 2016. Patcherex. https://github.com/angr/patcherex/.
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[7] Zuk Avraham. 2015. Experts Found a Unicorn in the Heart of Android. https://www.zimperium.com/blog/experts-found-a-unicorn-in-the-heart-of-android/.
[8] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2012. A Taint Based Approach for Smart Fuzzing. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. IEEE Computer Society. https://doi.org/10.1109/ICST.2012.182
[9] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2018.00046
[10] Code Intelligence Christian Holler. 2022. Dos and Don'ts when Introducing New Fuzzing Tools. https://www.code-intelligence.com/blog/implementing-fuzzing-tools
[11] Xpdf Community. 2024. Xpdf Fuzzing Harness Issue. https://github.com/google/oss-fuzz/issues/11711
[12] AFL++ developers. 2019. AFL++ instrumenting how to. https://aflplus.plus/docs/fuzzing_in_depth/.
[13] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2016.15
[14] Joshua Drake. 2015. Stagefright: Scary Code in the Heart of Android. (2015). Slideshow presented at Blackhat USA 2015.
[15] Joshua J. Drake. 2016. Stagefright: An Android Exploitation Case Study. USENIX Association.
[16] Will Drewry and Tavis Ormandy. 2007. Flayer: Exposing Application Internals. In *First USENIX Workshop on Offensive Technologies, WOOT '07, Boston, MA, USA, August 6, 2007*. USENIX Association. https://www.usenix.org/conference/woot-07/flayer-exposing-application-internals
[17] Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikström, and Martin Monperrus. 2022. Sorald: Automatic patch suggestions for sonarqube static analysis violations. *IEEE Transactions on Dependable and Secure Computing* (2022).
[18] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[19] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*.
[20] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3540250.3549098
[21] Gallopsled. 2013. pwntools. http://pwntools.com/.
[22] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE. https://doi.org/10.1109/ICSE.2009.5070546

[23] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013.* USENIX Association. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller

[24] haproxy community. 2023. haproxy Fuzzing Harness Issue. https://github.com/haproxy/haproxy/issues/2178#issuecomment-1584464122

[25] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* (nov 2020). https://doi.org/10.1145/3428334

[26] Marc Heuse. 2020. AFL++ LTO mode instrumentation. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md.

[27] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022.* ACM. https://doi.org/10.1145/3548606.3560575

[28] Ilias Kalouptsoglou, Miltiadis Siavvas, Dimitrios Tsoukalas, and Dionysios Kehagias. 2020. Cross-project vulnerability prediction based on software metrics and deep learning. In *Computational Science and Its Applications–ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part IV 20.* Springer.

[29] George Klees, 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* (New York, NY, USA). Association for Computing Machinery. https://doi.org/10.1145/3243734.3243804

[30] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[31] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.*

[32] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering.*

[34] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Communications of The Acm* (1990). https://doi.org/10.1145/96267.96279

[35] Balázs Mosolygó, Norbert Vándor, Gábor Antal, Péter Hegedűs, and Rudolf Ferenc. 2021. Towards a prototype based explainable javascript vulnerability prediction model. In *2021 International conference on code quality (ICCQ).* IEEE.

[36] NIST. [n. d.]. National Vulnerability Database. https://nvd.nist.gov/.

[37] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *23rd USENIX Security Symposium (USENIX Security 14).* USENIX Association, San Diego, CA. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng

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

[39] Pernosco. 2014. Record and Replay Framework. https://github.com/rr-debugger/rr.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[41] Kostya Serebryany. 2017. OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* USENIX Association. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[42] Alexey Smirnov and Tzi-cker Chiueh. 2005. DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA.* The Internet Society. https://www.ndss-symposium.org/ndss2005/dira-automatic-detection-identification-and-repair-control-hijacking-attacks/

[43] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18).* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3180155.3180250

[44] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/ramblr-making-reassembly-great-again/

[45] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA.* IEEE Computer Society. https://doi.org/10.1109/SP.2010.37

[46] Mark A Williams, Roberto Camacho Barranco, Sheikh Motahar Naim, Sumi Dey, M Shahriar Hossain, and Monika Akbar. 2020. A vulnerability analysis and prediction framework. *Computers & Security* 92 (2020), 101751.

[47] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Christopher Bookholt. 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005.* ACM, 223–234. https://doi.org/10.1145/1102120.1102151

[48] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[49] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597503.3639222