NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-43

Nov, 2025

Hardening the Fuzzing Ecosystem Through Automated Seed Generation, Report Deduplication, and Patch Correctness Validation

Yuhang Wu

Abstract

While fuzz testing has become a cornerstone of automated vulnerability discovery, its overall effectiveness is increasingly constrained by challenges that occur outside the fuzzing loop itself. This dissertation identifies three critical yet under-addressed components: seed preparation, bug report triage, and patch validation, as the true bottlenecks limiting the scalability and trustworthiness of fuzz-driven security analysis. I develop automated techniques for generating structurally valid and semantically rich seeds, significantly improving code coverage without manual effort. I further propose scalable, high-accuracy methods for deduplicating large volumes of fuzzer-generated bug reports, reducing triage delays and maintenance overhead. Finally, I introduce Klaus, a system that detects incorrect or incomplete kernel patches by analyzing their behavioral impact, uncovering previously unnoticed regressions and security-relevant inconsistencies. Together, these contributions strengthen the supporting pillars of the fuzzing lifecycle, ensuring that improvements in fuzzing throughput translate into meaningful, reliable advances in software vulnerability detection and remediation.

Keywords

Fuzz Testing; Seed Generation; Bug Report Deduplication; Patch Validation; Automated Vulnerability Discovery; Kernel Security; Program Analysis

NORTHWESTERN UNIVERSITY

Hardening the Fuzzing Ecosystem Through Automated Seed Generation, Report Deduplication, and Patch Correctness Validation

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By
Yuhang Wu

EVANSTON, ILLINOIS

December 2025

ABSTRACT

Hardening the Fuzzing Ecosystem Through Automated Seed Generation, Report Deduplication, and Patch Correctness Validation

Yuhang Wu

Fuzz testing has long served as a foundational methodology for discovering software vulnerabilities, enabling dynamic exploration of execution paths through structured yet randomized input mutations. Fuzz testing engines such as American Fuzzy Lop (AFL), LibFuzzer, and syzkaller have driven significant advances in automated vulnerability discovery and program state coverage. Despite these successes, recent evidence suggests that improving fuzzer mutation strategies, scheduling heuristics, or feedback metrics is no longer sufficient to strengthen the broader fuzzing centered software security ecosystem. A significant portion of the practical bottlenecks arises not within the act of fuzzing, but in the stages that precede and follow it.

This dissertation argues that three key components, seed preparation, bug report triage, and patch validation, constitute the true limiting factors for achieving scalable, trustworthy fuzz driven security analysis. These stages collectively determine whether increased fuzzing throughput can translate into effective vulnerability discovery and timely remediation.

First, the quality of input seeds plays a pivotal role in guiding fuzzers toward deep, semantically meaningful program states. Manually crafting such seeds is labor intensive and fundamentally non-scalable for complex software targets. I analyze the shortcomings of existing seed corpora and develop automated techniques to generate structurally valid, format-conscious, and semantically rich seeds tailored to program functionality. These techniques substantially improve code coverage and bug exposure while eliminating manual engineering overhead.

Second, the explosion in vulnerability reports generated by modern fuzzers has created severe

triage challenges. For example, syzkaller alone uncovered 3,736 Linux kernel bugs in only three years, yet nearly half of these reports were duplicates, imposing significant delays and resource burdens on maintainers. My work proposes more accurate, scalable mechanisms for deduplicating fuzz-generated bug reports, mitigating long backlogs, and enabling maintainers to focus on genuinely distinct and high impact issues.

Finally, the surge in discovered bugs naturally leads to a corresponding increase in patches. However, patch quality has not kept pace with the volume of patches. Approximately 6% of the Linux kernel patches are incorrect, with a subset introducing new security flaws or failing to fully address the underlying vulnerability. To address this, I design and implement Klaus, a system that detects incorrect or incomplete patches by analyzing patch behavior and its impact on system execution. Evaluated on real world kernel patches, Klaus identifies previously unnoticed regressions and security relevant inconsistencies, highlighting its practical value for maintaining the integrity of large, security-critical codebases.

Together, these contributions reinforce key operational pillars of the fuzzing lifecycle that lie beyond the design of the fuzzer itself. By strengthening seed generation, streamlining large scale bug triage, and safeguarding patch correctness, this work advances the reliability, scalability, and long term sustainability of fuzz-based software security analysis.

ACKNOWLEDGEMENTS

A journey of a thousand miles begins with a single step. Reaching this point in my Ph.D. and persisting through the journey is something I could not have imagined five years ago. I began learning computer science as an undergraduate, studying information security under the guidance of seniors and professors. It took me two years to step into the world of security competitions, and another year to become a core team member in international competitions. At that time, I thought I had reached the peak of my life. But by chance, I was given an opportunity to pursue a Ph.D., and at that moment, I realized that my path had only just begun.

These five years of doctoral study have felt even shorter than the two years I spent climbing from a beginner to the peak of security competitions. Not only did I have to push myself to explore cutting-edge technologies, but I also had to overcome the challenges of living abroad and navigating a foreign language. My days were filled with so many tasks that time seemed to pass even faster. Yet I must admit that the knowledge and life experience I gained, both academically and personally, are incomparable to anything I have experienced before.

I still remember my first year: with little preparation time, I needed to search for and read a tremendous number of papers while also keeping up with ongoing projects. My days were spent reading papers, analyzing vulnerabilities, and writing reports. It was difficult at first, but gradually, I learned how to read papers efficiently and debug programs swiftly. I owe much of this progress to the help of my advisor and senior lab mates.

In my second year, I followed my advisor to Northwestern University, which meant adapting to a new environment once again. Rent was higher, and the people around me were even more exceptional. I had to improve my English while adjusting to the fast-paced quarter system. I still remember overloading myself with courses one quarter, falling behind on research, and even

experiencing tension with my advisor. But by restructuring my time and pushing myself, I still managed to complete everything on schedule. That experience taught me how to remain calm under pressure and plan my work with confidence.

I remember the joy of having our first paper accepted, as well as the nervousness of preparing slides and giving my first talk. An amusing memory was when PowerPoint crashed during my USENIX presentation; I had to reopen the file and locate the right slide, wasting a precious minute. Still, I completed the presentation calmly. I also cherish every sincere conversation with my advisor; our relationship often felt more like that of friends.

In the past year, I participated with my lab and collaborators from other universities in DARPA's AIxCC competition. We achieved excellent results in the semifinals. Unfortunately, differing visions within the team made collaboration difficult, and I eventually withdrew. I still care deeply about their progress and hope their work becomes meaningful and impactful. As graduation approaches, I sometimes feel regret and guilt—if I had stayed, would the outcome have been even better?

As memories surge back, I want to begin by expressing my deepest gratitude to my advisor, Professor Xinyu Xing, for accompanying me through these five years with patience and understanding. Whenever I was lost, he always offered timely guidance. He supported me in both life and research. The plans we discussed five years ago have gradually come true, and I hope he will continue to make contributions that benefit society and humanity.

I would also like to thank the guy who first guided me toward the Ph.D. path, Dr. Zhenpeng Lin. We met through security competitions, came to trust each other through repeated collaboration, and during a critical turning point in my life, he encouraged me to pursue a Ph.D., giving me new direction and motivation. He provided tremendous help throughout my Ph.D. journey.

Next, I want to thank Dr. Dongliang Mu, Dr. Yueqi Chen, and Dr. Gang Wang, who led me

through the publication of my first paper. You were my academic torchbearers, and I am deeply grateful for your patience and guidance. I also want to thank my lab seniors and colleagues: Dr. Wenbo Guo, Dr. Xian Wu, Qi Qin, Dr. Jiahao Yu, and Dr. Zheng Yu, for your collaboration and your support in both research and daily life. I believe each of us will shine in our respective paths in the future.

I am grateful to my parents for their unconditional support. I am proud and fortunate to have such open-minded parents who never stopped me from studying abroad and never allowed me to struggle financially. My achievements are inseparable from your love and nurturing. I also thank my friends in China and my teammates from security competitions, who provided emotional support during my lonely moments abroad and stood with me through countless competitions.

Finally, I want to extend my sincere thanks to my committee members: Professor Xinyu Xing, Professor Yan Chen, Professor Gang Wang, and Professor Yueqi Chen, for your invaluable advice during my defense. It is an honor to have you witness my graduation.

There are many more people I wish to thank, those who accompanied me, supported me, and helped me grow. I will carry your kindness forward, striving to contribute to society and to pass this kindness on.

THESIS COMMITTEE

Xinyu Xing
Northwestern University
Committee Chair

Yan Chen
Northwestern University
Committee Member

Xiao Wang
Northwestern University
Committee Member

Yueqi Chen
University of Colorado Boulder
Committee Member

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Fuzz testing, commonly referred to as fuzzing, has become one of the most important techniques in modern software security. At its core, fuzzing is conceptually simple: a program is repeatedly executed on large volumes of mutated inputs, known as test cases or seeds, while sanitizers monitor for crashes and anomalous behavior [1]. Despite this simple philosophy, fuzzing has proven remarkably powerful. Because it is automated, scalable, and general, fuzzing has been successfully applied across domains ranging from user-space applications to kernels and embedded systems. Over the past decade, it has evolved from a research idea into a foundational pillar of the software security ecosystem.

This evolution is reflected in the diversity and maturity of today's fuzzers. General-purpose fuzzers such as AFL [2] and AFL++ [3] remain widely adopted for user-space software; libFuzzer [4] enables efficient in-process fuzzing of APIs, and syzkaller [5] has become the standard for fuzzing the Linux kernel. Beyond individual tools, major organizations now operate fully automated security ecosystems built around fuzzing. Google's OSS-Fuzz [6], for example, orchestrates massive cloud infrastructure to fuzz over one thousand open-source projects continuously. Powered by the ClusterFuzz framework [7], OSS-Fuzz has already uncovered tens of thousands of vulnerabilities and bugs with minimal human intervention. Similarly, syzbot, the automated platform built on top of syzkaller, continuously discovers, reports, and tracks bugs in the Linux kernel [8], coordinating efforts between researchers and maintainers.

These ecosystems demonstrate a clear trend: fuzzing is no longer just a tool but the centerpiece of an increasingly automated vulnerability-discovery pipeline. A typical pipeline now includes (1)

preparing project-specific seeds; (2) running fuzzers at scale; (3) triaging large volumes of bug reports; and (4) generating and validating patches. As automation expands, the ecosystem depends on the smooth operation of every stage. However, the rapid growth in scale has surfaced several critical bottlenecks that directly limit efficiency and reliability.

The first challenge lies in the pre-fuzzing stage. Modern software often expects highly structured or complex inputs. Without high-quality seeds, the fuzzer may struggle to reach deep program states, reducing coverage and slowing vulnerability discovery [9]. Yet, seed preparation for new projects today often requires domain knowledge, manual reading of documentation, and significant human effort, making it difficult to scale to the thousands of projects targeted by automated systems.

A second challenge emerges after fuzzing begins. Large fuzzing deployments routinely generate tens of thousands of bug reports. While sanitizers provide basic information such as stack traces and crash types, understanding the true root cause often requires detailed static and dynamic analysis. Worse, many reports are duplicates: superficially different symptoms caused by the same underlying vulnerability. Prior analyses show that duplicate reports can account for nearly half of all fuzz-discovered bugs [10]. This duplication leads to wasted developer effort, patch delays, and bottlenecks in vulnerability-response workflows.

Finally, the ecosystem faces increasing pressure in the post-patch stage. Once developers propose a fix, a crucial but often overlooked question remains: is the patch correct? Incorrect or incomplete patches can leave systems vulnerable, introduce regressions, or trigger new security issues [11]. Yet automated methods for validating patch correctness remain limited, especially for complex systems like the Linux kernel.

The dissertation is organized as follows.

Chapter 2 presents an automated seed-generation framework developed as part of our submis-

sion to the DARPA AI Cyber Challenge (AIxCC) [12]. Modern fuzzing pipelines rely heavily on high-quality seeds to explore deep program states efficiently, yet generating such seeds typically requires extensive manual analysis of input specifications, file structures, and API contracts. This manual step forms a critical bottleneck that prevents large-scale fuzzing ecosystems from achieving full automation.

In this work, we design and implement a fully automated framework that harnesses large language models to infer a program's expected input format, synthesize diverse and structurally valid seeds, and adapt to new projects with minimal human intervention. The system integrates seamlessly with existing fuzzers and enables scalable, project-agnostic seed preparation. Evaluations across a wide range of real-world open-source targets demonstrate that our automatically generated seeds improve both coverage and fuzzing efficiency compared to traditional handcrafted seeds. The techniques introduced in this chapter formed the core of our finalist submission to the DARPA AIxCC Competition [13].

Chapter 3 presents a comprehensive empirical study on the duplication of bug reports in the Linux kernel fuzzing ecosystem. Continuous fuzzing systems such as Syzkaller and Syzbot have achieved remarkable success over the past several years, discovering more kernel vulnerabilities than had been identified in the previous two decades. However, this success also introduces a significant side effect: continuous fuzzing produces an enormous volume of crash reports, and many of these reports are in fact duplicates generated by the same underlying bug. Although Syzbot employs a simple heuristic to group these reports, our observations show that this mechanism is often inaccurate and can misrepresent the true number of unique bugs.

In this work, we aim to understand the nature and impact of this duplication problem. We collected all fixed Linux kernel bugs reported by Syzbot from September 2017 to November 2020, amounting to 3.24 million crash reports grouped under 2,526 bug entries. Our analysis reveals

that duplication is highly prevalent: 47.1% of these bug entries correspond to the same underlying vulnerability as one or more other entries. By examining associated metadata, such as timestamps, discussions, and developer responses, we show that undetected duplication introduces additional costs, including longer resolution times and increased developer workload.

To gain deeper insight into the causes of duplication, we conducted a detailed investigation of a representative sample consisting of 375 crash reports covering 120 distinct kernel bugs. With the assistance of experienced Linux kernel developers, we identified six key contributing factors that lead to duplicated reports, ranging from differing triggering contexts to variations in sanitizer output and execution paths.

Guided by these empirical findings, we designed and prototyped several practical deduplication strategies aimed at improving the accuracy of grouping fuzzing-generated crash reports. We evaluated these strategies using a ground-truth dataset and demonstrated their effectiveness in reducing duplication. Applying these improved techniques to open Syzbot bugs allowed us to uncover previously unrecognized duplication cases that had remained unnoticed in the existing reporting pipeline.

Chapter 3 appeared in *the Proceedings of the Network and Distributed System Security Symposium (NDSS)* [14].

Chapter 4 presents a new methodology for evaluating the correctness of Linux kernel patches. As the Linux kernel continues to evolve and incorporate new features, bugs are introduced daily. While many of these issues can be detected and resolved with the assistance of automated analyzers and fuzzing systems, creating correct patches remains a persistent challenge. Incorrect or incomplete patches can introduce new vulnerabilities, leave root causes unaddressed, or even enable severe security exploits. Despite the importance of patch correctness, systematic tools for validating kernel patches remain limited.

To better understand the nature of incorrect patches, we first conduct a detailed manual analysis of 182 real-world Linux kernel patches that were later found to be flawed. This analysis reveals a recurring pattern: many incorrect patches inadvertently modify a program's read or write operations in ways that violate intended semantics. These alterations often manifest only in specific execution contexts, making them difficult to detect with existing tools.

Building on these observations, we develop KLAUS, a principled and automated method for evaluating patch correctness. The system uses abstract interpretation to identify the read and write operations affected by a patch and then incorporates these alterations into a branch-guided fuzzing mechanism. By directing execution toward the code regions and contexts most influenced by the patch, KLAUS is able to expose semantic inconsistencies and detect incorrect fixes more effectively than general-purpose fuzzing or static analysis alone.

We evaluate KLAUS on hundreds of real-world Linux patches and demonstrate that it outperforms existing approaches in both effectiveness and efficiency. To date, KLAUS has identified and reported numerous incorrect patches, including those capable of enabling privilege escalation on modern Android and Ubuntu systems, highlighting the practical impact and necessity of automated patch evaluation.

Chapter 4 is to appear in *the Proceedings of the USENIX Security Symposium 2023* [15].

Chapter 5 concludes the dissertation and discusses future research directions.

CHAPTER 2
AUTOMATED SEED GENERATION VIA LLM-GUIDED INPUT FORMAT INFERENCE

2.1 Introduction

Modern software increasingly relies on complex and highly structured input formats [16]. Many applications, such as image parsers, compilers, browsers, and state-dependent systems, perform strict input validation early in execution [17]. For fuzzers, this poses a significant challenge: randomly mutated inputs rarely satisfy these constraints, preventing the fuzzer from progressing past the entry point and severely limiting its ability to explore deeper program logic [2].

Prior efforts have attempted to address this issue through comparison-hooking techniques [18] or manually crafted grammars [19], [20]. However, comparison hooks provide only localized feedback and fail in scenarios such as hash-based validation, while manual grammar construction fundamentally breaks the goal of scalable, automated fuzzing. As a result, current approaches cannot automatically infer the global structure of a program's input format, an essential requirement for enabling deeper fuzzing [21].

In this chapter, we explore how large language models (LLMs) can be used to automatically understand and model a program's expected input structure [22]. By analyzing the code responsible for input parsing and validation, an LLM can synthesize a grammar describing the format of valid inputs, enabling automatic generation of structured seeds that allow fuzzers to bypass early checks [23].

We design a fully automated framework that identifies program entry points and input sources [24],

extracts validation-relevant code via static analysis [25], and uses an LLM agent to infer a concise and expressive grammar. This grammar is then used to generate high-quality seeds through a grammar-based mutator [26], enabling effective fuzzing without any manual intervention.

To summarize, this chapter makes the following contributions:

- We introduce a fully automated methodology that leverages large language models to infer input formats and generate grammars for structured seed synthesis.
- We design and implement a complete seed-generation pipeline, including entry-point discovery, validation-logic extraction, LLM-driven grammar inference, and grammar-based seed generation, that integrates seamlessly with existing fuzzers.
- We demonstrate the practicality and effectiveness of this approach through evaluations on real-world Java vulnerabilities and in the DARPA AIxCC competition [27], where our framework enabled early and successful bug discovery.

2.2 Motivating Example

Fuzzers rely on generating valid or near-valid inputs to progress beyond early parsing logic. However, modern programs often contain complex validation routines that reject malformed inputs immediately. Using only random mutations, a fuzzer frequently becomes stuck at the entry point, unable to explore deeper code or trigger meaningful behaviors.

Listing 1 shows a concrete example from the DARPA AIxCC Java challenge. The program expects inputs following the structure `headerKey \x00 headerValue \x00 command`. To validate the request, the program computes the SHA-256 hash of a fixed string, `breakin the law`, and checks whether the request contains the special header `x-evil-backdoor`. If the header exists, its value is hashed and compared against the target hash.

```java
public void doexecCommandUtils(Request request, ...) {

    // program input format: headerKey \ headerValue \ command
    byte[] sha256 = DigestUtils.sha256("breakin the law");

    if (containsHeader(request.getHeaderNames(), "x-evil-backdoor")) {
        String backdoorValue = request.getHeader("x-evil-backdoor");

        // hash the user-supplied header value
        byte[] providedHash = DigestUtils.sha256(backdoorValue);

        // The hash of headerValue must equal the hash of "breakin the law"
        // A valid input example:
        //   x-evil-backdoor \ breakin the law \ whoami
        if (MessageDigest.isEqual(sha256, providedHash)) {
            // trigger the bug
        }
    } else {
        new Event(Event.Status.ERROR,
                  "Error: Only Admin Users Are Permitted", cmdSeq2);
    }
}
```

Listing 1: Java code snippet from the AIxCC challenge. Triggering the bug requires satisfying a hash-level comparison, which traditional fuzzers cannot reason about or mutate toward.

The key observation is that this comparison occurs at the hash level, not the string level. Traditional fuzzers attempt to guide mutations by hooking string comparisons (e.g., `memcmp`) and using them as feedback. But in this case, by the time the comparison is executed, the original string is already transformed into a 32-byte digest. The comparison routine only sees opaque byte arrays, providing no usable feedback to the fuzzer. As a result, the fuzzer cannot infer how to mutate the input to satisfy the validation.

A human analyst, however, can examine the code and immediately recognize that any header value equal to `breakin the law` will satisfy the hash check. With a crafted input such as `x-evil-backdoor \x00 breakin the law \x00 whoami`, the vulnerable logic is reached immediately.

This example highlights a fundamental limitation of existing fuzzing techniques: without understanding the semantic structure of input-validation logic, fuzzers cannot navigate past complex

checks, especially when they involve hashing, cryptographic transformations, or multi-stage conditions. This motivates the need for an automated methodology, powered by large language models, that can infer input formats, understand validation semantics, and synthesize structurally valid seeds capable of driving the fuzzer deeper into the program.

2.3 Overview

Modern fuzzing pipelines depend critically on the quality of their initial seeds. When targeting programs with complex or deeply structured input formats, traditional mutation-based fuzzers frequently fail to progress beyond early parsing logic, resulting in shallow coverage and limited vulnerability discovery. To address this challenge, we design a fully automated framework that extracts input-validation logic directly from real-world software, infers the underlying input structure using a combination of static analysis and LLM-based reasoning, and generates high-quality structured seeds capable of driving the fuzzer deeper into the target program. Figure 2.1 presents an overview of the framework and its main components.

At a high level, the system begins by applying a modified compiler to the target codebase. Unlike a conventional compilation pipeline, our compiler is instrumented at key function boundaries and dataflow points. The purpose of this instrumentation is twofold. First, it enables the system to dynamically observe the execution paths exercised during initial runs of the program, allowing it to automatically identify program entry points as well as all user-controlled input sources. Second, the instrumentation provides a foundation for tracking how input values propagate through the program during execution, which later guides the extraction of code segments responsible for input parsing, transformation, and validation.

Once the program has been instrumented, the framework performs static analysis to pinpoint the code that governs input validation. Many real-world applications contain multiple layers of

parsing code, nested conditional logic, and scattered validation routines embedded across utility functions, middleware components, and library calls. Naively providing the entire source tree to a large language model would exceed context limitations and drastically reduce analysis quality. To circumvent this problem, our framework employs a modular static analysis phase that isolates only those functions, control-flow regions, and data dependencies that directly interact with user-controlled input. These extracted regions form a compact but semantically complete slice of the program's validation logic.

Building on this foundation, the system invokes an LLM-driven agent to infer the structural constraints required by the program's input format. Unlike direct prompting, which would require manually curated context, our agent operates autonomously and interacts with a local code search engine. The agent retrieves code on demand, issuing search queries when additional semantic context is needed and selectively expanding its view of the program only when necessary. The agent therefore behaves similarly to a human analyst: it inspects relevant parts of the codebase, reconstructs the expected format of valid inputs, and converts this understanding into a formal grammar specification. This grammar captures not only the syntactic structure (e.g., delimiters, token positions, production rules) but also semantic constraints implied by the program logic, such as value dependencies, ordering requirements, or relationships between multiple fields.

With the grammar synthesized, the system proceeds to seed generation. Our framework builds upon the design of Grammar Mutator [28], which accepts grammar files as input and produces structurally valid seed templates. However, merely generating valid seeds is insufficient for effective fuzzing; seeds must also be placed under a mutation engine that can preserve structural integrity while introducing diversity. To achieve this, we implement a modified version of LibFuzzer that incorporates a grammar-aware mutation strategy. This extended fuzzer parses the grammar tree directly and applies specialized mutation operators that respect the grammar while exploring

variations that can trigger deeper execution paths. For example, the mutator can substitute alternative productions within non-terminals, expand or contract recursive structures, or adjust token-level values while maintaining overall input validity.

The integration of LLM-based inference, static analysis, and grammar-driven fuzzing forms a tightly unified pipeline. Each stage provides essential information to the next: instrumentation reveals input sources; static analysis exposes validation logic; the LLM agent infers the input structure; and the grammar-based mutator converts that structure into actionable fuzzing inputs. This modular but interconnected design allows the system to generalize across diverse software projects without manual tuning. More importantly, it enables the fuzzer to bypass early parsing and validation barriers that would otherwise prevent exploration of deeper program states. Through this pipeline, the system transforms the inherently hard problem of input-format inference into an automated, scalable, and reproducible process capable of supporting large-scale fuzzing deployments.

2.4 Technical Details

In this section, we present the technical foundations of our automated seed-generation framework. We begin by describing how our modified compiler instruments the target program and captures the information necessary to identify entry points and user-controlled input sources. We then detail our static-analysis pipeline, which isolates the code segments responsible for input parsing and validation, forming the basis for subsequent grammar inference. Finally, we discuss how our LLM-based agent synthesizes a grammar from the extracted validation logic and how this grammar is transformed into structured seeds through a grammar-aware mutation engine.

At the core of our approach is a combination of compiler instrumentation, selective code extraction, and LLM-guided reasoning. Our modified compiler plays a crucial role in exposing dynamic behaviors of the target program, enabling the system to automatically trace how inputs propagate

across function boundaries. Building on this dynamic information, the static-analysis phase identifies the relevant input-handling logic and prepares a compact representation suitable for analysis by the LLM agent. This section explains how each component is orchestrated to infer complete input grammars and produce high-quality seeds without human intervention.

2.4.1 Compiler Instrumentation

A central requirement of our framework is the ability to automatically determine (1) where external input first enters the program and (2) which functions participate in processing this input. To support this, we extend the `javac` compiler with a custom instrumentation pass that traces the flow of user-controlled data from the moment it is read from the fixed input file to all program locations that depend on it.

Identifying User-Controlled Input Sources. In our setting, all user-controlled data originates from a predetermined file that the Java program loads at runtime. During compilation, our modified `javac` scans the abstract syntax tree (AST) for file-access operations such as: `new FileInputStream(...)`,`new FileReader(...)`,`BufferedReader.readLine()`, and any wrapper utilities around these primitives. Whenever such operations are encountered, the compiler automatically injects logging instrumentation before the data is consumed. These probes capture: ❶ The concrete source of the input (i.e., the fixed file name). ❷ The program location (class, method, and line number). ❸ The raw string or byte sequence that was read. This inserted instrumentation allows us to treat each file-read operation as a synthetic input source, and all values derived from these reads as user-controlled.

Tracing Input Flow and Discovering Entry Points. Input sources alone are insufficient; we must also identify the functions where meaningful input processing begins, that is, the entry points for our grammar-extraction analysis. To achieve this, the instrumentation also records the

caller and callee relationships of the functions that operate on the read data. Specifically: When an instrumented input value is passed as an argument to another method, when it is stored in a field and later retrieved, or when it is used in parsing logic (e.g., `split()`, `substring()`, `parseInt()`, regex matches), the runtime trace logs these dependency edges. After execution, the system constructs a dynamic call graph rooted at the input source. The first set of methods that directly process the input, e.g., functions that parse tokens, validate headers, dispatch commands, or extract fields, are identified as entry points. These entry points form the boundary where our grammar inference begins: they define the initial functions responsible for interpreting the structure of the input file.

2.4.2 Static Analysis

Following the compiler instrumentation phase, the next step of our pipeline is to isolate the code responsible for parsing and validating user-controlled input. To accomplish this, we perform a targeted static analysis over the Java bytecode of the instrumented program, focusing specifically on instructions whose operands may be derived from the input sources identified during dynamic execution.

Our analysis begins by marking all bytecode values that originate from the traced input operations as tainted. This includes not only the immediate results of file-read methods (e.g., `readLine()`, `read()`, `readBytes()`), but also all values derived from these inputs through data-flow propagation. For example, intermediate substrings, tokenized components, numeric conversions, and wrapper objects are all treated as taint carriers.

With taint propagation in place, the analysis scans the program's bytecode for operations that compare, transform, or otherwise constrain tainted values. In particular, we track:

- string and byte comparisons (e.g., `String.equals()`, `startsWith()`).

Input: Bytecode B of the target program; set S of input-source variables traced at runtime
Output: Set V of validation-relevant code blocks
$V \leftarrow \emptyset$
$W \leftarrow S$
foreach $x \in S$ **do**
  mark x as tainted
end
while W *is not empty* **do**
  $x \leftarrow$ pop an element from W
  foreach *instruction* $inst$ *in* B *that uses* x **do**
    if $inst$ *performs comparison or validation on* x **then**
      $block \leftarrow \mathsf{BasicBlock}(inst)$
      $V \leftarrow V \cup \{block\}$
    end
    $y \leftarrow$ result of $inst$
    if y *is not tainted* **then**
      mark y as tainted
      push y into W
    end
  end
end
return V

Algorithm 1: Static Analysis for Input-Validation Logic

- arithmetic or relational checks on tainted integer values or string lengths.
- hash computations on tainted data (e.g., `MessageDigest.digest()`).
- branching instructions whose predicates depend on tainted values (e.g., `ifcmp`, `ifnull`, `tableswitch`).
- parsing utilities that impose format constraints (e.g., `split()`, `matches()`, `parseInt()`, regex-based parsers).

Each of these operations represents a semantic condition that valid inputs must satisfy. Whenever such an operation is encountered, we extract the surrounding control-flow region—typically

the basic block or method containing the comparison, and add it to a growing set of validation-relevant code segments. Because Java bytecode preserves precise control-flow boundaries, our analysis can capture entire conditional structures, including fall-through blocks and alternative branches associated with tainted predicates.

By iteratively tracing taint propagation and collecting all blocks that constrain input-derived values, the analysis produces a compact, self-contained slice of the program's input-handling logic. This slice reflects the complete set of syntactic and semantic checks that the program applies before deeper execution can occur. These extracted code segments form the basis for the grammar inference stage: they provide the LLM agent with precisely the code required to understand the expected input structure, without exposing it to unrelated implementation details or overwhelming the model with unnecessary context.

2.4.3 LLM Agent for Context-Aware Code Understanding

After extracting the bytecode regions involved in input parsing and validation, the next stage of our pipeline is to interpret these regions and infer the structural constraints that govern the program's expected input format. Importantly, directly providing the entire project codebase to an LLM is neither feasible nor effective: it would exceed typical context limits and substantially degrade the model's reasoning quality. To avoid this, we adopt an agent-based design in which an LLM interacts with a local code search engine built on top of an open-source framework OpenGrok [29].

The core idea is to expose only the code that is strictly relevant to input-validation logic, while granting the agent the ability to retrieve additional context on demand. At the beginning of this stage, the agent receives the set of validation-relevant bytecode blocks produced by our static analysis. These blocks form a compact slice of the program that captures the key conditions, comparisons, and semantic checks that inputs must satisfy. However, the validation logic often

depends on surrounding helper functions, class methods, or utility routines that lie outside this slice.

To bridge this gap, the agent issues targeted queries to the local code search engine whenever it encounters a reference whose behavior is not fully visible in the current context. Queries may be based on method names, signatures, callsites, class types, or associated control-flow regions. The search engine, indexed offline using the underlying OpenGrok infrastructure, efficiently returns a concise and relevant subset of the code. The agent then integrates these results into its working state, allowing it to expand its understanding incrementally and only when necessary. This avoids the token overhead and semantic noise associated with providing the full project source to the model.

Through this selective retrieval process, the LLM agent operates much like a human code analyst: it begins with the core validation logic, follows dependencies that influence input structure, retrieves additional definitions or helper routines as needed, and constructs a holistic view of the expected input format. Once the agent has recovered the complete set of syntactic and semantic constraints implied by the validated code paths, it synthesizes a formal grammar describing the structure of valid inputs. This grammar captures mandatory fields, token orderings, delimiter rules, structural alternatives, and value-dependent constraints derived from the program logic.

By combining LLM reasoning with an efficient search-based retrieval mechanism, this stage ensures that the model receives exactly the information necessary for accurate grammar inference, no more and no less. This not only reduces token usage and improves inference efficiency, but also significantly enhances the accuracy and consistency of the grammars generated for real-world software systems.

2.4.4 Grammar-Aware Seed Generation and Mutation

After the LLM agent synthesizes a grammar that captures the structural and semantic rules of the target program's input format, our system operationalizes this grammar to support both high-quality seed generation and grammar-preserving mutation during fuzzing. To produce the initial seed corpus, we leverage the existing Grammar Mutator [28] framework, widely used in AFL for grammar-based input generation. Since the grammar produced by our LLM is translated into the JSON format expected by Grammar Mutator, the tool can be used directly—without modification—to generate a diverse set of syntactically valid seeds. These seeds already conform to the structural constraints inferred from the program's input-validation logic and can be fed directly into Jazzer [30] as the starting corpus for fuzzing.

While Grammar Mutator's seed generator can be used off-the-shelf, its mutation engine is tightly integrated with AFL's native fuzzing pipeline and therefore cannot be directly applied to the JVM-based environment targeted by Jazzer. To enable grammar-aware fuzzing in this setting, we design and implement a new mutation subsystem within Jazzer that recreates the essential capabilities of Grammar Mutator's structure-preserving mutation logic. Our modified Jazzer parses the grammar specification at startup, constructs an internal representation of the production rules, and maps mutation operators to the corresponding grammar nodes. During fuzzing, these operators perform guided mutations such as replacing alternative productions, expanding or contracting recursive elements, permuting list structures, and modifying literal fields while preserving global grammatical validity.

2.5 Implementation

We implemented our fully automated seed-generation and grammar-aware fuzzing framework for Java applications by extending three major components: (1) a modified Java compiler for input-source tracing, (2) a bytecode-level static-analysis pipeline, and (3) a grammar-based fuzzing back-end that integrates Grammar Mutator with an enhanced version of Jazzer. In this section, we describe key aspects of our implementation.

Modified Java Compiler. Our system begins by instrumenting the target application to identify entry points and user-controlled input sources during dynamic execution. We modify the OpenJDK `javac` frontend to inject lightweight tracing instructions at method boundaries and I/O-related APIs. Since the Java compiler normally discards intermediate representations, we added an additional pass to preserve annotated AST nodes and bytecode-level metadata required by our later analysis. During execution, any instruction that loads data originating from the designated input file or stream is tagged as tainted; the compiler records these taint origins in an auxiliary metadata file that we consume during static analysis.

This design avoids modifying the JVM or introducing heavy instrumentation. Because the instrumentation is limited to high-level compiler constructs, it remains stable across Java versions and introduces negligible runtime overhead.

Static Analysis on Java Bytecode. The bytecode-level static analysis stage reconstructs the validation logic that constrains the program's input format. Using the taint origins provided by the compiler, we perform a backward slice over the program's bytecode to identify all instructions whose semantics depend on tainted values. This includes string comparisons (`String.equals`, `startsWith`, `regionMatches`), array comparisons (`Arrays.equals`), arithmetic and relational checks on tainted integers, cryptographic and hash computations, regular expression match-

ing, and control-flow predicates (`ifcmp`, `ifnull`, `tableswitch`) whose outcomes influence parsing behavior.

To localize the relevant logic, we identify the enclosing basic blocks for each tainted comparison and merge adjacent blocks into larger code regions representing input-validation paths. These extracted regions form the minimal slice of the program that encodes its input grammar. By operating directly on JVM bytecode, our analysis does not depend on source availability and naturally handles optimized or obfuscated builds.

LLM Agent and Code Search Engine Integration. The extracted validation slice represents only the core constraints, but many checks depend on helper methods or type definitions scattered across the project. To retrieve these dependencies, we integrate an autonomous LLM agent with a local code search engine based on OpenGrok [29]. The search engine indexes the entire project, enabling efficient retrieval of method bodies, type hierarchies, and callsite neighborhoods.

During analysis, the agent selectively queries the search engine whenever it encounters an unresolved reference in the validation slice. The retrieved code is incrementally incorporated into the agent's working context, allowing it to reconstruct the complete semantic rules governing valid inputs without overloading the LLM with the full project codebase. This on-demand retrieval significantly reduces token usage and improves grammar inference accuracy.

Grammar-based Seed Generation. Once the agent synthesizes a grammar describing the program's expected input format, we translate this grammar into the JSON specification used by Grammar Mutator, an existing AFL++ component for grammar-driven input generation [28]. Grammar Mutator is used directly—without modification—to generate an initial set of well-formed, structurally diverse seed inputs. These seeds immediately satisfy the program's syntactic constraints and can be supplied to any fuzzing engine. In our implementation, these grammar-derived seeds form the starting corpus for Jazzer.

Integrating Grammar Mutator with Jazzer. While Grammar Mutator's seed generator can be used off-the-shelf, its mutation logic is tightly coupled with AFL++ and cannot be directly applied to JVM fuzzing. To support grammar-preserving fuzzing in Jazzer's libFuzzer-style environment, we implemented a new mutation subsystem inside Jazzer.

At startup, the modified Jazzer parses the grammar file and constructs an internal grammar tree. We then attach custom mutation operators to each production rule. These operators perform structure-preserving transformations such as substituting alternative productions, adjusting list lengths, modifying recursive repetitions, or mutating literal fields while maintaining global grammatical validity. These mutations are seamlessly integrated into Jazzer's mutation pipeline and are invoked in place of standard byte-level mutations when grammar information is available.

2.6 Evaluation

We evaluate our system using both real-world Java vulnerabilities and a practical deployment during the DARPA AI Cyber Challenge (AIxCC) semifinal. Our study seeks to answer two questions: (1) whether LLM-generated grammars accelerate bug discovery in Java programs, and (2) whether grammar-aware fuzzing enables Jazzer to reach deeper program states that baseline fuzzing cannot.

2.6.1 Dataset and Experimental Setup

We constructed a ground-truth dataset of ten real-world Java vulnerabilities spanning deserialization vulnerabilities, server-side request forgery (SSRF), and sandbox bypasses. All vulnerabilities include publicly documented triggering test cases, allowing us to determine whether the fuzzer successfully reaches the vulnerable state.

Each target was analyzed using our full pipeline: compiler instrumentation, bytecode-level static analysis, LLM-driven grammar inference, Grammar Mutator seed generation, and our mod-

Table 2.1: Evaluation on Ten Real-World Java Vulnerabilities

Vulnerability (CVE)	Category	Triggered by Our System?
CVE-2015-8103	Deserialization RCE	✓
CVE-2016-0792	Deserialization RCE	✓
CVE-2018-1000182	SSRF (Git Plugin)	✓
CVE-2019-1003000	Groovy Sandbox Bypass	✓
CVE-2019-1003026	SSRF (Mattermost Plugin)	✓
CVE-2020-2121	Yaml Deserialization RCE	✓
CVE-2017-1000353	Deserialization RCE	✗
CVE-2019-1003001	Groovy Sandbox Bypass	✗
CVE-2019-1003028	SSRF (JMS Plugin)	✗
CVE-2020-2166	Yaml Deserialization RCE	✗

ified Jazzer with grammar-aware mutation. For comparison, we also ran baseline Jazzer with its default byte-level mutation and no structured seeds. All fuzzing executions were run for a fixed two-minute window across identical hardware and configuration.

Table 2.1 summarizes the vulnerabilities and our results.

2.6.2 Bug Discovery Performance

Our system successfully triggered six out of the ten vulnerabilities within two minutes of fuzzing, demonstrating a substantial improvement over baseline Jazzer. In all six successful cases, Jazzer with LLM-generated structured seeds reached the vulnerable code path almost immediately, whereas baseline Jazzer failed to do so within the same time window.

These results reaffirm the importance of well-structured initial seeds in Java fuzzing. For targets with strict input-validation logic, such as those using XStream, SnakeYAML, or plugin-specific configuration parsers, baseline fuzzers rarely generate syntactically correct inputs early enough to explore deeper states. In contrast, the grammar-informed seeds produced by our system bypass these early checks and quickly drive the program to complex parsing routines where vulnerabilities

are located.

2.6.3 Impact in AIxCC

Beyond controlled evaluation, our system demonstrated strong real-world effectiveness during the DARPA AI Cyber Challenge (AIxCC) semifinal. Applied to a large and complex Java project included in the challenge suite, our framework enabled our team to be the first among all semifinalists to discover a previously unknown vulnerability. The high-quality grammar-derived seeds played a decisive role in this early discovery, underscoring the practical impact of our approach in competitive and operational settings.

2.7 Related Work

2.7.1 Program Analysis for Inferring Input Structure

Several systems leverage program analysis to reconstruct input formats or validation logic. Tupni [31] recovers structural constraints via dynamic taint tracking and dataflow analysis. Polyglot [32] performs multi-language input extraction by analyzing string operations along tainted execution paths. AUTOGRAM [33] automatically derives grammars from execution traces by identifying parsing-related code segments. More recently, DIGGER [34] combines symbolic execution and grammar extraction to recover hierarchical input structure from complex parsers. Our approach differs in that it operates primarily on Java bytecode and selectively extracts validation-relevant logic through a combination of static slicing and code search, without requiring full dynamic coverage.

2.7.2 Grammar-Based Input Generation and Mutation

Grammar-based generation has long been recognized as an effective technique for exploring programs that process structured inputs. LangFuzz [35] and Nautilus [36] use user-specified gram-

mars to generate syntactically valid test cases. Superion [37] extends mutation-based fuzzers with grammar-aware transformations for XML and JavaScript. Grammar Mutator [28], originally designed for AFL++, provides grammar-driven seed generation and mutation but targets native binaries. Our system reuses Grammar Mutator for seed generation but implements a new JVM-compatible mutation backend for Jazzer, enabling grammar-preserving fuzzing for Java applications.

2.7.3 LLMs for Code Understanding and Structural Reasoning

Recent advances in large language models have shown strong capabilities in recovering semantic structure from source code. Codex [38] and CodeBERT [39] demonstrate that LLMs can infer program intent, dependency structure, and API usage patterns. LLM-assisted fuzzing approaches such as [40], [41] generate test cases using language models, but typically produce raw strings without explicit structural guarantees. In contrast, our approach uses LLMs to infer grammars' explicit, machine-consumable representations of input structure—directly from a program's validation logic, enabling systematic, grammar-aware fuzzing.

2.8 Conclusion

In this work, we introduced an automated framework that combines program analysis and LLM-driven grammar inference to generate high-quality, structurally valid seeds for Java fuzzing. By extracting validation logic from bytecode, synthesizing input grammars, and integrating grammar-based seed generation with a customized grammar-aware mutation engine in Jazzer, our system enables significantly deeper exploration of Java applications than traditional byte-level fuzzing. Our evaluation on real-world CVEs and our success in the AIxCC semifinal demonstrate that grammar-informed seeds can dramatically accelerate bug discovery and reach complex parsing

logic quickly. This work shows that LLM-guided grammar inference is a practical and effective foundation for improving fuzzing efficiency in structured-input domains.

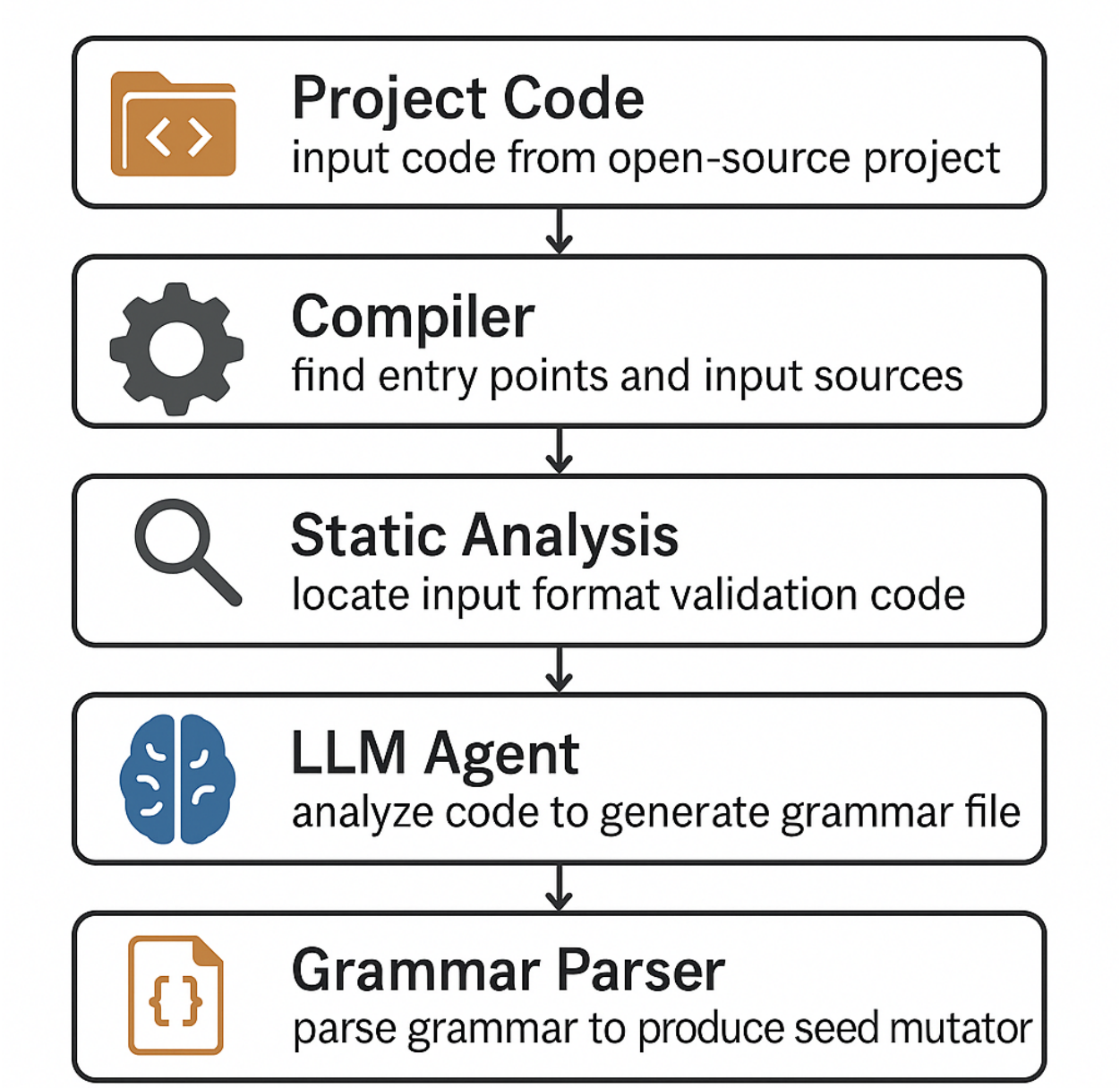

Figure 2.1: Overview of our automated seed-generation framework. Given the source code of an open-source project, the system first analyzes the program through a customized compiler to automatically identify entry points and input sources. It then performs static analysis to locate the code responsible for input parsing and validation, isolating the logic that governs the program's expected input structure. Building on this extracted information, an LLM-based agent infers a grammar that captures the required format of valid inputs. Finally, the generated grammar is processed by a grammar-based parser, which produces structured seeds that enable the fuzzer to bypass early validation checks and explore deeper program behaviors.

CHAPTER 3

AN IN-DEPTH ANALYSIS OF DUPLICATED LINUX KERNEL BUG REPORTS

3.1 Introduction

In this chapter, we focus on the problem of duplicated bug reports in operating system kernels. The kernel is the most critical component of an OS, and its security directly affects the security of both the system and all user applications. Kernel vulnerabilities are widely considered more severe than those in user-space programs [42], and have been repeatedly exploited in major real-world attacks such as WannaCry [43], DirtyCow [44], and BleedingTooth [45]. As large-scale fuzzing infrastructures (e.g., syzkaller and syzbot) continuously discover new kernel bugs, they also generate massive numbers of crash reports, many of which are duplicates caused by the same underlying vulnerability. In this section, we analyze this deduplication problem in depth, quantify its impact on kernel development, and propose practical strategies to reduce duplicate kernel bug reports.

To proactively detect and patch kernel vulnerabilities, the security community has investigated significant efforts. These efforts include both developing more advanced fuzzing tools to detect new vulnerabilities [46], [47], [48] and organizing security analysts and kernel developers to analyze the reported bugs and develop patches.

The progress of kernel bug detection has been slow in the past 20 years until recently when Google initialized Syzkaller [5] and Syzbot [8] projects in 2016. These are open-source projects that automatically and continuously fuzz main Linux kernel branches to find bugs. In addition to the advanced fuzzing techniques (Syzkaller), another key advantage is that the system (Syzbot)

produces standard crash reports and structured information fields (e.g., vulnerable kernel versions, kennel configurations), which makes it easier for security analysts to reproduce bugs and analyze root causes. These efforts have been vastly successful. As of November 2020, the system has found 3,736 kernel bugs in just three years, which is more than the total number of kernel bugs identified in *the past 20 years before Syzkaller* [49].

Bug Report Duplication Problem. While continual fuzzing of Syzbot has significantly improved the efficiency of kernel bug discovery, it also produces an excessive amount of crash reports. In the past three years, Syzbot has generated over 10 million crash reports, the vast majority of which are "duplicated", meaning that the crashes are triggered by the same bugs. Considering that security analysts need to *manually* analyze these reports to assess the severity of the bug and pinpoint the root cause, it is highly desirable to group the crash reports caused by the same bug together. Currently, Syzbot relies on a simple heuristic to perform deduplication: if the crashes share the same *crash function* and *crash type*, then they will be grouped under the same bug report, sharing the same bug title.

Unfortunately, this heuristic-based deduplication method is not accurate. Anecdotally on Syzbot dashboard, we have observed that certain crash reports caused by the same bug were not successfully grouped together (because they have different crash functions or crash types). As a result, the bug reports about the same bug were treated as new (different) bugs, and then were assigned to different analysts. On one hand, multiple groups of analysts working on the same bug in parallel without communicating with each other leads to inefficiency (i.e., redundant manual effort). On the other hand, a limited view of the diverse bug behaviors of the same bug may lead to incomplete patches [50].

Goals and Approaches. In this paper, we empirically analyze the duplicated kernel bug reports from Syzbot. We define duplicated bug reports as those that *share the same root cause*. Our goal is

to understand: (1) the prevalence of duplication; (2) potential costs introduced by duplication; and (3) the causing factors to the report duplication. Based on the insights obtained from the empirical analysis, we further explore low-cost solutions for bug deduplication.

A key challenge in our study is to obtain the "ground-truth" for bug report duplication. While this challenge is difficult to resolve for open bugs that are currently being analyzed by developers, we could group bugs that are already fixed. The idea is to link bug reports based on their "patches" — if multiple bug reports are fixed/closed by the same patch, these bugs are highly likely to be the same bug. After this initial grouping, we then manually analyze the bugs to confirm duplication based on root causes. In this work, we have collected *all* of the fixed kernel bugs reported on Syzbot from September 2017 to November 2020. This includes 2,526 bug reports and over 3.24 million crash reports. Based on their patches, we group these bug reports into 1,686 unique kernel bug groups. Out of the 2,526 bug reports, 1,191 (47.1%) are duplicated with one or more other reports.

We answer the first two questions by analyzing the ground-truth bug groups. We observe that kernel bugs already take a long time to fix, even *without duplication* (the median open time is 70 days). Bug groups *with duplication* take an even longer time to close. Even after the first bug in the group is fixed, the other duplicated bugs will remain open for extra time, consuming valuable resources (e.g., developers' time). Also, not too surprisingly, bug groups with duplication involve more developers in the bug fixing process.

To answer the third question, the most effective approach is to manually analyze these kernel bugs. To ensure the reliability of results, we have organized security experts and developed rigorous analysis procedures (including peer-reviews). We select 120 bug groups that Syzbot failed to detect the duplication (120/351 = 34.2%). Under the guidance of an approved IRB protocol, 5 Linux kernel experts are organized to set up the environments, reproduce the reported bugs, and

analyze the code changes and the developer's notes to figure out the causes of bug duplication. In total, we collectively identify 6 main contributing factors include different inputs, thread interleaving, memory dynamics, kernel versions and branches, different sanitizers, and inline function.

Based on the empirical findings, we propose and prototype five actionable strategies for bug deduplication during reporting time. For instance, we enhance the existing memory quarantine mechanism and replace the slab allocator with the slub allocator in the tested kernel to eliminate the influence of memory layout dynamics. Then we swap and run the PoC programs from potentially duplicated bug reports to observe their behaviors (i.e., mitigating the influence of kernel implementation and function inline). We inject delay to the kernel source code and run the PoC programs multiple times to fully expose the possible thread interleaving. To rule out the influence of different sanitizers, we run the PoC programs with the same sanitizer configuration. Finally, we compute the similarity of PoC programs using a customized Levenshtein distance to handle the different behaviors caused by input differences.

We evaluate these strategies with our ground-truth dataset and show that they can effectively identify duplicated bug report pairs, with a true positive rate of 80% and a false positive rate of 0.01%. Among the proposed techniques, the technique that handles "different inputs" is the only source of false positives, and the other four techniques are false-positive-free by design. We further apply our techniques to the *real-world open bugs* and confirm that they can identify *previously unknown* bug duplication cases. From both fixed and open bugs, we have found cases where the developers were misled to produce incomplete patches due to a limited view of the diverse bug behaviors.

Contributions. In summary, we have three key contributions:

- First, we empirically analyze duplicated reports of Linux kernel bugs, and identify the limitations of existing deduplication heuristics. We show that the undetected duplication introduces extra

costs (time and developer efforts) and even produces incorrect patches.

- Second, we organize Linux kernel experts to perform an in-depth analysis of duplicated bug reports. We identify six key contributing factors to duplication.

- Third, we propose and prototype a series of strategies to alleviate the bug duplication problem. We evaluate them against both ground-truth data and current open bugs to demonstrate their effectiveness. To facilitate future works, we will release our code and dataset with this paper.

Our work provides new insights into the causing factors of kernel bug deduplication, and introduces an initial solution. We have shared our results and findings with the Syzkaller and Syzbot teams (and some kernel developers), and have received positive feedback. In the end of the paper, we discuss the open challenges to kernel bug deduplication. We believe further research is needed in order to fully address the problem.

3.2 Background and Motivation

In this section, we describe the background of kernel fuzzing and bug reports, and introduce Syzkaller and Syzbot. Then we describe our problem setup and research goals.

Syzkaller: Kernel Fuzzing to Detect Bugs. To harden the security of kernel programs, the security community has developed fuzzing tools to discover kernel bugs **Syzkaller**, [46], [47], [48]. Among existing fuzzers, Syzkaller **Syzkaller** is by far the most successful efforts *in practice*. Syzkaller is an open-source project initiated by Google in 2016 (popularized in 2017). As of November 2020, Syzkaller has found 3,736 kernel bugs (2,526 of them are now patched) in just three years, which is more than the total number of kernel bugs identified in *the past 20 years before Syzkaller [49]*.

Syzkaller has several advanced designs. First, it leverages a declarative description of syscall interface to manipulate programs (sequences of syscalls), and uses code coverage feedback as guidance to explore all the kernel code space. Second, syzkaller leverages the fault injection mechanism [51] in Linux kernel to inject failures (*e.g.,* allocation failures) into the runtime execution of system calls. After a kernel bug is found, Syzkaller will try to generate (and minimize) syz and C reproducers. Finally, Syzkaller coordinates with many different sanitizers (*e.g.,* KASAN [52], [53], KMSAN [54], [55], KCSAN [56], KUBSAN [57]), kernel detection mechanisms (*e.g.,* KMEMLEAK [58],ODEBUG [59]) and other pre-defined assertions (*e.g.,* BUG_ON, WARN_ON) to detect kernel vulnerabilities at runtime. These detection tools allow syzkaller to expose all mainstream security bugs such as memory error bugs (*e.g.,* Memory Leak, Null Pointer Dereference, Use-After-Free (UAF), pre-defined assertion (*e.g.,* WARN, BUG), deadlocks and concurrency bugs. Therefore, Syzkaller can cover a highly diverse set of bugs and bug types.

Syzbot: Continuous Kernel Fuzzing and Reporting. For a long time (before Syzkaller), running kernel fuzzers and reporting bugs have been almost exclusively manual efforts. The lack of automation and bug reporting standards has created significant difficulty in bug reproduction and patching [60]. To automate bug discovery and reporting, the Syzkaller team further developed a *continuous* fuzzing system for kernel programs called *Syzbot* [8]. Syzbot system continuously and automatically updates and fuzzes main Linux kernel branches (*e.g.,* upstream, linux-stable) with different Syzkaller instances. Once bugs are found, they will be *automatically* reported to corresponding kernel developers with standardized information (e.g., crash reports). Analysts from the kernel community will first analyze the bug (manually) to confirm its validity. Then kernel developers will analyze the root cause of the bug, and develop a patch to fix the bug.

To facilitate bug analysis and information sharing, Syzbot provides an open forum (called "Syzbot dashboard") to list and keep track of the reported kernel bugs. Each bug has its own web

page that contains key information about the bug, such as the vulnerable kernel versions, the kernel configuration file, the Syzkaller repositories, reproducer (syz repro or C repro). Both syz repro and C repro are PoC (Proof-of-Concept) files to reproduce the crash. The configuration file shows which sanitizers are enabled in the corresponding kernel crash. Finally, it is worth noticing that fixed bugs all have a "Fix commit" which is the kernel commit (patch) that has fixed the underlying bug.

A key side effect of continuous fuzzing is it generates a large number of inputs to trigger bugs, which produces an excessive number of crash reports. In recent three years, Syzbot has produced over 10 million crash reports, many of which were actually triggered by the same bug. Such a duplication level could negatively impact the efficiency of kernel developers who need to analyze the reported bugs *manually*. Currently, Syzbot follows a simple heuristic to group (or deduplicate) crash reports. If the crashes appear at the same *function* and share the same *crash type*, then these crash reports will be grouped together, under the same *bug title*. For example, under the bug title "`KASAN: use-after-free Read in map_lookup_elem`", all the crash reports share the same crash function (i.e., `map_lookup_elem`) and the crash type (i.e., "`KASAN: use-after-free Read`").

Limitations of the Current Deduplication Method. The current deduplication method is coarse-grained and error-prone. Anecdotally on Syzbot dashboard, we observed certain crash reports caused by the same bug were not successfully grouped under the same *bug titles*. Instead, they are treated as distinct bugs and are assigned to different analysts. Such "duplicated" open bugs lead to concerning problems. First, multiple groups of analysts working on the same bug in parallel without communicating is an inefficient way of using the analysts' time. Second, once one of the duplicated bugs is fixed, there will be extra delays to close other bugs that share the same root causes (i.e., wasting analysts' time if they keep working on them). Third, without grouping these

bugs, analysts do not have the complete view of the bug behaviors, which can lead to incomplete and incorrect patches. We discovered real cases which will be presented later.

For these reasons, we want to empirically understand the bug report duplication problem on Syzbot, and answer the following questions. First, how prevalent is bug report duplication on Syzbot? Does duplication indeed introduce extra costs to analyzing and patching the bug (Section 3.3–3.4)? Second, what are the main causes to the report duplcation (Section 3.5)? Third, how can we effectively deduplicate kernel bug reports (Section 3.6)?

3.3 Methodology and Dataset

In this section, we first define our problem scope and then describe the collected dataset to measure the prevalence of bug report duplication. Finally, we describe our workflow to identify the causes of the duplication.

3.3.1 Method Overview

Definition of the Root Cause of a Bug. The root cause of a bug is defined as the faulty code leading the Linux kernel into an abnormal state. Take the kernel bug `#bbeb6e43` [61] as an example. The root cause of this bug is in the function `array_map_alloc`. If the `attr->max_entries` field goes beyond a threshold (i.e., `0xfffffffd`), an integer overflow will occur in the variable `array_size`, causing `array_map_alloc` to allocate an oversized buffer. The oversized buffer eventually leads to a general page fault (GPF) or an Out-of-Bound (OOB) memory access.

Definition of Unique Bug. A bug is uniquely defined by its root cause. In other words, if multiple crash reports share the same root cause, then we define them as duplicated reports. At the high level, our idea is to collect historical crash reports of Linux kernels and link reports that share the

same root cause (*i.e.,* reports of the same bug). Based on the linked reports, we develop an analysis procedure to systematically examine the reasons for duplication.

We first define the key terms used in this paper. In Figure A.3 in the Appendix, we use an example to show the hierarchical relationships between bug groups, bug titles, and crash reports. When Syzbot reports a crash, it automatically generates a *bug title* formed by a crash function and crash type. For example, the title "`UBSAN: shift-out-of-bounds in mceusb_dev_recv`" means the crash happens on function "`mceusb_dev_recv`" and the crash type is "`UBSAN: shift-out-of-bounds`". Under continuous fuzzing, it is common for the same bug to be triggered multiple times since a bug would remain unfixed a certain amount of time. Syzbot currently groups these crash reports under the same *bug title*. In the example of Figure A.3, Bug Title B has N crash reports under the same title.

As mentioned before, grouping crash reports using crash function and crash type is often inaccurate, because the same bug may exhibit different crash behaviors (i.e., with different crash functions or crash types). In the example of Figure A.3, Bug Titles A, B, C are in fact triggered by the same bug and thus should have been grouped under the same *bug group*. Here, the bug group represents the "ground-truth" unique bug.

3.3.2 Problem Scope

The current crash deduplication method is coarse-grained and error-prone, manifesting two undesired outcomes: 1) false positives (FP)—bug reports with different root causes, grouped into the same title; and 2) false negatives (FN)—bug reports with the same root cause are not grouped under the same title.

By design, Syzbot can handle the false positive problem. Given a false positive case (i.e., bug reports with different root causes, grouped into the same title), kernel developers may only

fix one of them during their first attempt, leaving the others unfixed. However, as a continuous fuzzing system, Syzbot will continue to fuzz the patched version and keep filing crash reports for the unfixed bugs. According to Syzbot developers [62], the fact that the new crash reports have the same title as the fixed one is an indication that there are other bugs unfixed, and thus developers will continue to work on it. To this end, the falsely grouped bugs will not be missed[1]. For this reason, our paper will focus on "false negative" cases (*i.e.,* bug report duplication), which can lead to duplicated effort by kernel developers and reduced efficiency.

Furthermore, we also do not consider the case where one crash reported by Syzkaller is triggered by multiple bugs. In practice, this situation is extremely rare. To have a crash tied to multiple bugs, two conditions must be satisfied: 1) the input to trigger multiple bugs need to be carefully crafted; 2) no sanitizers nor internal detection mechanisms are enabled during the fuzzing process, which will allow an error state to propagate sufficiently far away from the bug-triggering site. However, Syzkaller generates random inputs based on the code coverage feedback during kernel fuzzing, and it enables all kinds of detection mechanisms mentioned in Section 3.2 to find the bug at its first appearance. As such, it is safe to rule out such cases.

Challenges. Our first challenge is to link and verify duplicated reports for the same bug to establish "ground-truth". Second, to understand the reasons behind the report duplication, we need to extensively analyze the crash behaviors (for different kernel subsystems). This process, unfortunately, is difficult to fully automate and is time-consuming. Third, even for manual analysis, kernel bug analysis requires a high level of domain expertise. As such, it is difficult to simply crowdsource the analytic tasks (e.g., via Amazon Mechanical Turk).

Approaches. With these challenges in mind, we consider the following strategies. First, instead

[1] As a concrete example, "memory leak in hub_event" [63] contains multiple memory leak bugs in different kernel drivers, grouped under the same bug title. After Syzbot assigns the patch for one of the bugs, we observe that Syzbot still continues to generate this bug report [64] during fuzzing since not all bugs are fixed.

Category	Crash Reports	Bug Titles	GT Bug Groups
Fixed Bugs	3,243,946	2,526	1,686
Duplicated	803,206	1,191	351
Sampled	90,519	375	120

Table 3.1: Dataset overview. "GT Bug Group" refers to the number of ground-truth bug groups.

of analyzing the "open" bugs, we focus on the historical kernel bugs that have been patched by developers. By analyzing these patches, we can potentially, in turn, link the bug reports caused by the same bug (that were not successfully grouped during reporting). We will further analyze the grouped bug reports manually to confirm their root causes and establish the "ground-truth". Second, considering the time-consuming nature of bug analysis, we prioritize the *depth of the analysis* while maintaining a reasonable coverage for generalizable results. We randomly sample a set of bugs that have different crash behaviors (i.e., with duplicated reports) and form a focused group of domain experts to work on deduplication experiments. Based on our results, we will propose light-weighted solutions to link bug reports automatically.

3.3.3 Kernel Bug Report Dataset

To support our analysis, we collect kernel bug reports from Syzbot dashboard. Syzbot dashboard provides more complete and update-to-date bug reports (compared with CVE [65]). More importantly, Syzbot dashboard has included the necessary information (*e.g.,* kernel version, kernel configuration, Proof-of-Concept) needed for bug reproduction (which are often missing on the CVE site [60]).

On the Syzbot dashboard, there are three main queues for kernel bugs, namely, "Open bugs", "Fixed bugs", and "Invalid bugs". As of November 2020, there are 8,071 bugs with more than 10 million crash reports in total in the three queues. As we discussed above, only the bugs in "Fixed bug" queue can be linked to establish the ground-truth for our analysis. The reason is the fixed bugs

contain one additional field called "Fix commit" which shows the patch that fixes the underlying kernel bug. Reported crashes that are eventually fixed by the same patch are highly likely to share the same root cause[2] (i.e., the same bug). In the following, we leverage the patch information as a proxy to group duplicated bugs, and will further validate the ground-truth manually in Section 3.5.

Our Dataset. Focusing on "Fixed bugs", we crawl the entire queue from Syzbot dashboard. Our dataset summarized in Table 3.1 covers all the corresponding crash reports from September 24, 2017, to November 11, 2020. In total, there are 3,243,946 crash reports grouped under 2,526 bug titles. As mentioned, the bug title only groups crash reports based on the crash function and crash type, which could be highly inaccurate.

Then we analyze the patches of the bugs and identify 1,686 "ground-truth" bug groups. If a bug group contains more than one unique bug title, we regard the bug group as having *duplication*.

Note that, in theory, this grouping method could incorrectly group bug reports if *one patch is used to fix multiple bugs*. However, our manual analysis on these bug groups (see Section 3.5) has confirmed that there was no such case, i.e., one patch is always used to fix one bug. This is expected since "one patch per bug" is a policy that the Linux kernel community has been enforcing before pushing a patch to the kernel [66]. As such, our ground-truth is valid.

As shown in Table 3.1, we find in total 351 bug groups that have duplication (20.8%), involving 1,191 unique bug titles (47.1%) and 803,206 crash reports (24.8%). In other words, in these groups, multiple bug titles and their crash reports were failed to be linked together during the time of bug reporting and patch development. As a result, developers might have wasted time on analyzing the duplicated crash reports (i.e., incorrectly treating them as new bugs).

Sampling Set. To understand the causes of duplication, we organize domain experts to analyze the

[2]Using the patch to group crash reports is a reliable method except for rare cases, e.g., the incorrect patch was assigned due to human errors. We will discuss such rare cases in detail in Appendix-A.

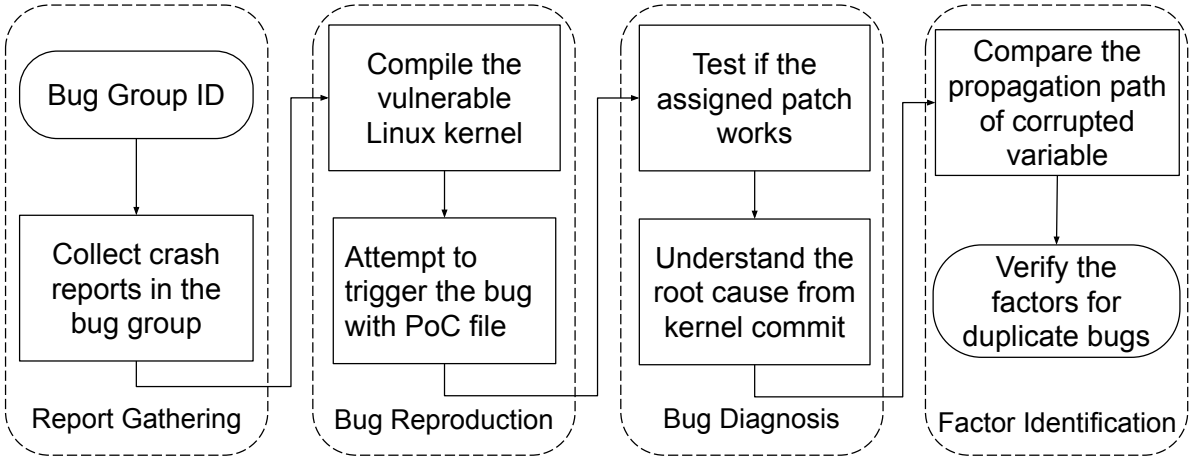

Figure 3.1: Workflow to analyze the contributing factors to different crash behaviors

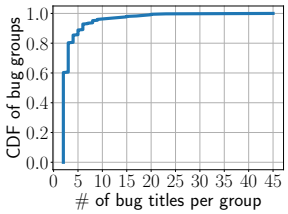

Figure 3.2: Number of bug titles for each bug group with duplication.

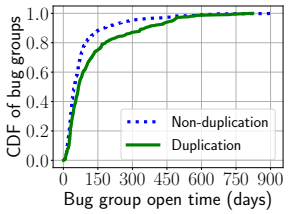

Figure 3.3: Open time of each bug group: duplicated vs. non-duplicated.

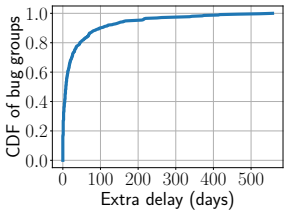

Figure 3.4: Extra delay introduced by duplication.

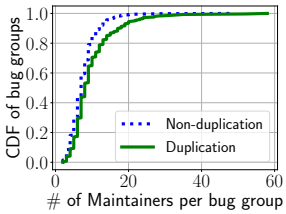

Figure 3.5: Number of maintainers per bug group.

reported bugs manually. As mentioned, we prioritize the depth of analysis, and sample a subset of bug groups that has duplication. Our sampling is not completely random, but has two preferences. *First*, we prioritize analyzing bugs that cover more *diverse* crash behaviors (*i.e.,* based on the crash type and crash function in the bug title). *Second*, we prioritize analyzing bugs that have more *severe* crash behaviors. For example, Out-Of-Bound, Use-After-Free, Invalid-Free are considered to be more severe bugs [67], [68].

From the 351 bug groups that contain duplication, we sampled 120 bug groups (34.2%) which covers 375 bug titles and 90,519 crash reports. As shown in Figure A.4 (in Appendix), this sampled set covers diverse crash types.

Justifications on the Dataset Size. We believe our dataset of 120 unique kernel bugs is reasonably large for our purposes. On one hand, the 120 bugs already cover 34.2% of the bug groups that have duplication. On the other hand, this dataset is already several times larger than existing datasets used by previous works for kernel bug analysis. For example, after a literature review, we find that most of the used kernel bug datasets contain fewer than 20 bugs [69], [70], [71], [72], [73], [74], [75], [76]. Several works studied 20-60 bugs [77], [78], [79], [80], but they are not focusing on bug reproduction and root cause analysis. Instead, most of them focus on analyzing the code changes in the patches that can be easily automated. A related work [81] collected 373 CVEs to verify the generated hot patches (for Android kernels), and only used 3 working exploits to verify the correctness of generated patches.

3.3.4 Experiment Design

We design an experiment to examine the reasons that manifested different crash behaviors for the same bug (i.e., the cause of the duplication in bug reports). Our study was reviewed and approved by our IRB(#STUDY00008566).

Crash Deduplication Workflow. The workflow is shown in Figure 3.1. (1) Given a bug group, a security analyst collects all the crash reports and related files (including PoCs). (2) The analyst downloads and compiles the vulnerable Linux kernel based on the kernel commit and configuration file, and then runs the vulnerable Linux kernel in QEMU [82]. The analyst reproduces the bug with the provided PoC. (3) The analyst tests the patch and examines the root cause by reading the commit message and code changes. (4) The analyst identifies the corrupted variables based on the root cause and then compares the propagation path of corrupted variables from the root cause to the crashing site. The analyst then verifies the factors that contribute to the different crash behaviors.

In the above process, manual analysis is needed for (2)–(4). This is because, with the fuzzing log alone, it is difficult to identify the root causes and infer the contributing factors to the diverse crash behaviors.

The Analyst Team. We have formed a strong team of 5 security analysts to carry out this experiment. Each analyst has not only in-depth knowledge of the different subsystems in the Linux kernel but also has first-hand experience analyzing Linux kernel bugs, writing exploits, and developing patches. The analysts regularly publish at top security venues and have rich CTF (Capture-the-Flag) experience. When one security analyst finishes the analysis of one bug group, this analyst will present the details of this bug and explain the identified factors to the other four analysts for a peer review. After all the analysts confirm the correctness of the results, we then close the case for the given bug group.

3.4 Result: Impact of Duplication

In this section, we analyze the crash report dataset and illustrate the general impact of duplication on bug fixing time and developer overhead. Later in Section 3.5, we will focus on the reasons behind the duplication in our expert-driven analysis.

We first analyze the crash reports and bug groups in our dataset (Table 3.1). Recall that hundreds or thousands of crash reports are automatically grouped under a bug title (defined by crash function and crash type) by Syzbot. However, the same bug may lead to different crash functions or crash types, and thus have multiple bug titles (i.e., duplication). In the following, we characterize the duplicated bug titles and analyze the costs introduced by duplication.

Duplication Level. Figure 3.2 shows the number of distinct bug titles per bug group in our duplication set (351 bug groups in total). We find that about 60% of bug groups have only two bug titles

and the duplication level is not very high. However, about 14% of the bug groups have more than five bug titles. The largest bug group has 45 distinct bug titles. This indicates the same bug can cause highly diverse crash behaviors (i.e., different crash functions or crash types).

Extra Delay due to Duplication. We suspect that bug groups with duplication would take a longer time to close (i.e., having a longer open time) compared to those without duplication.

Figure 3.3 confirms our hypothesis. More specifically, given a bug group, we define its *open time* based on two timestamps: t_1 (the date when the first crash was reported); and t_2 (the date when the last bug title in this bug group was closed). The open time of this bug group is simply $t_2 - t_1$. Figure 3.3 shows two main results. First, kernel bugs, even without duplication, have a longer open time. The median open time for kernel bugs without duplication is around 70 days (more than two months). Among them, about 10% take more than 150 days (5 months) to close. Some bugs take as long as 900 days (2.4 years). Second, bug groups with duplication indeed take a longer time to be closed. For example, for bug groups with duplication, about 20% are open for more than 150 days.

In Figure 3.4, we explicitly plot the extra delay introduced by the duplication. For bug groups with duplication, if all their bug titles were immediately linked/grouped during the time of reporting, then all of these bug titles should have been closed immediately when *the first bug title* was successfully patched and closed. However, in practice, these bug titles were not grouped automatically, and thus some bug titles remain open even though the bug is already fixed. We calculate the *extra delay* as the time gap between the first bug title closing time and the last bug title closing time in the same bug group. We find that for 80% of the bug groups, the extra delay is within 50 days. However, a small portion (5%) of the bug groups have an extra delay of more than 200 days. The extra delays could be harmful, i.e., wasting the maintainers' time if they still treated the bugs as open bugs and kept working on them.

Ideally, once a bug is fixed, the remaining bug titles in the same group can be recognized (automatically) since these bugs will no longer be triggered after the patch. In practice, there are possible reasons for their delayed closing. First, some bug titles do not have any reproducer, and thus they cannot be tested automatically to recognize that they are already fixed. Second, some bug titles need to be closed by maintainers manually with a `#syz-fix` tag, and thus experience delays.

Less-Optimized Manual Efforts. Another potential cost of duplication is the inefficient coordination of maintainers. For example, different groups of maintainers may independently spend time analyzing the same bug (under different bug titles) without knowing each others' work. This could lead to redundant efforts of maintainers or even incomplete patches (see cases in Section 3.5.3). For each of the fixed bugs in Table 3.1, we attempted to obtain the list of maintainers. While this information is not directly available at the Syzbot dashboard, most bug reports have a google group where maintainers discuss the bug. We obtain the complete maintainer lists for 82.3% of the bug groups with duplication and 80.8% of the groups without duplication. Using this data, we plot Figure 3.5, which confirms that bug groups with duplication often have more maintainers than non-duplicated bug groups.

For bug groups with duplication, we further analyze how much the maintainer lists under different bug titles overlap. Given a bug group, each bug has its own set of maintainers. We compute an *overlap ratio* as the size of the intersection of these maintainer sets over the size of their union. We find that 16.6% of the bug groups have an overlap ratio of 0, which means the duplicated bug reports are handled by completely different groups of maintainers. The average and median ratio is 0.56 and 0.62 respectively. If these bugs were deduplicated upon reporting, it would be possible to assign them to the same set of maintainers to avoid redundant efforts.

Category	Bug Titles	Bug Groups
Sampled Set	375	120
Reproduced	339	109
Final Ground-truth	327	104

Table 3.2: Summary of manually analyzed dataset.

3.5 Results: Contributing Factors

Next, to understand the reasons for the duplication (i.e., different crash behaviors for the same bug), we have performed expert-driven analysis following workflow described in Section 3.3.4.

3.5.1 Manual Crash Analysis

This analysis is focused on the sampled set that includes 120 bug groups and 90,519 crash reports under 375 bug titles (see Table 3.1). Among these crash reports, 2,873 of them have included the PoC files needed for bug reproduction. The analysis tasks took 5 security analysts about 2,400 man-hours to finish. On average, each kernel bug took 8 hours to complete all the proposed steps. Based on our experience, the most time-consuming part is to understand the root cause and identify the propagation path of corrupted variables from the root cause to the crashing site.

Out of those 120 bugs, we successfully reproduced and identified the root causes for 109 bug groups (see Table 3.2). The other bugs were not reproducible, and thus we could not proceed with the rest of the analysis. Also, among the 109 bug groups, we find 5 bug groups for which the Syzkaller team has incorrectly assigned the patch (see more details in Appendix-A). Therefore, we use the remaining 104 bug groups as the final set to report our findings on contributing factors to duplication.

Factors	Bug Groups	Percentage
Different Inputs	55	50.5%
Thread Interleaving	18	16.5%
Memory Dynamics	18	16.5%
Kernel Versions and Branches	14	12.8%
Different Sanitizers	13	11.9%
Inline Function	8	7.3%

Table 3.3: The number of bug groups that are affected by each factor. One bug group could be affected by multiple factors.

3.5.2 Reasons for Duplication

We identify 6 factors leading to bug report duplication. We summarize these factors in Table 3.3.

Factor 1: Different Inputs. The first reason for duplicated bug titles is the *input difference*. Given a kernel bug and the corresponding buggy code snippet, there could be *various execution contexts* and *many different paths* towards the buggy code. Following these paths under different contexts, the bug might demonstrate different errors and thus lead to different types of crash reports.

Take the bug `#bbeb6e43` [61] and its two reports [83], [84] as an example. The input programs extracted from the reports have the exact same sequence of system calls but one difference in the argument. Both programs could interact with kernel code and trigger the bug. However, they reach the buggy code through slightly different execution paths, which stop the kernel execution at two different kernel functions and demonstrate different types of kernel errors. As a result, the crash reports cannot be grouped together by the current deduplication method.

In addition to the *different paths*, differences in *execution contexts* could also lead to duplicated crash reports. Take, for example, the kernel code shown in Table 3.4. The function

```
1  int copy_verifier_state(...) {
2    struct bpf_func_state *dst;
3    dst = kzalloc(sizeof( *dst), GFP_KERNEL);
4    if (!dst)
5      return -ENOMEM;
6    return 0;
7  }
8
9  int is_state_visited(...) {
10   struct bpf_verifier_state_list *new_sl;
11   copy_verifier_state(&new_sl->state, cur);
12   // no error handler if allocation fails
13   free_verifier_state(&new_sl->state, false);
14 }
15
16 int push_stack(struct bpf_verifier_env *env, ...) {
17   struct bpf_verifier_stack_elem *elem;
18   err = copy_verifier_state(&elem->st, cur);
19   // no error handler if allocation fails
20   pop_stack(env, NULL, NULL);
21 }
```

Table 3.4: The code snippet showing that injecting allocation fault at different contexts could cause different bug reporting results.

`copy_verifier_state` will throw an error code `ENOMEM` if the allocation function `kzalloc` fails. Since the buggy kernel does not handle this error correctly, when the function `copy_verifier_state` returns, the kernel will experience a GPF. Table 3.4 shows that there are two sites calling the function `copy_verifier_state` (i.e., line 11 and 18). As such, when a kernel fuzzer injects the allocation error at different calling contexts (at the line 11 and 18 respectively), the kernel reports GPF in different kernel functions (i.e., `free_verifier_state` and `pop_stack`), resulting in two different crash reports.

Out of the 109 bug groups, we find 55 bug groups (50.5%) are affected by the input difference (involving 185 distinct bug titles and 88,456 crash reports). This is the most prevalent cause of the duplication problem (see Table 3.3).

Factor 2: Thread Interleaving. Linux kernel is an asynchronous system supporting multi-task mechanisms. Incorrect synchronization (or missing synchronization) between kernel threads not only introduces concurrency bugs [85], but may also produce different types of errors (i.e., leading to report duplication).

```
1  // Thread A
2  void put_pi_state(... pi_state) {
3    if (atomic_dec_and_test(&pi_state->refcount) {
4      kfree(pi_state);
5    }
6  }
7
8  // Thread B
9  void exit_pi_state_list(... curr) {
10   struct list_head *h = &curr->pi_state_list;
11   struct futex_pi_state pi_state = list_first_entry(h);
12   lock(&pi_state->pi_mutex.wait_lock);
13   get_pi_state(pi_state);
14 }
15
16 void get_pi_state(... pi_state) {
17   WARN_ON_ONCE(!atomic_inc_not_zero(&pi_state->refcount));
18 }
```

Table 3.5: The code snippet illustrating the difference in thread interleaving could cause duplicated bug reporting.

To illustrate this problem, Table 3.5 shows an example where `pi_state` is a variable shared between thread-`A` and thread-`B`. Using the same input program but through different thread interleaving, the kernel could experience different errors. For example, thread-`A` invokes the function `put_pi_state`, decreasing the reference count for `pi_state` to zero (Line 3). Following the reference count being set to zero, thread-`B` calls the function `get_pi_state` at the Line 16. This function examines whether the reference count for `pi_state` is zero. If the condition holds, it reports a warning at Line 17. In addition to this thread-interleaving, another synchronization between both threads is to call the function `put_pi_state` in thread-`A` prior to thread-`B` executing Line 12. In this thread scheduling, thread-`A` de-allocates the object `struct futex_pi_state`, leaving behind a dangling pointer `pi_state`. At Line 12, thread-`B` dereferences the dangling pointer and a use-after-free error occurs.

Among all the kernel bugs we inspected, we discover that thread-interleaving has affected 16.5% of the bug groups that contain duplicated bug titles. It is the second most prevalent factor that contributes to bug report duplication (see Table 3.3).

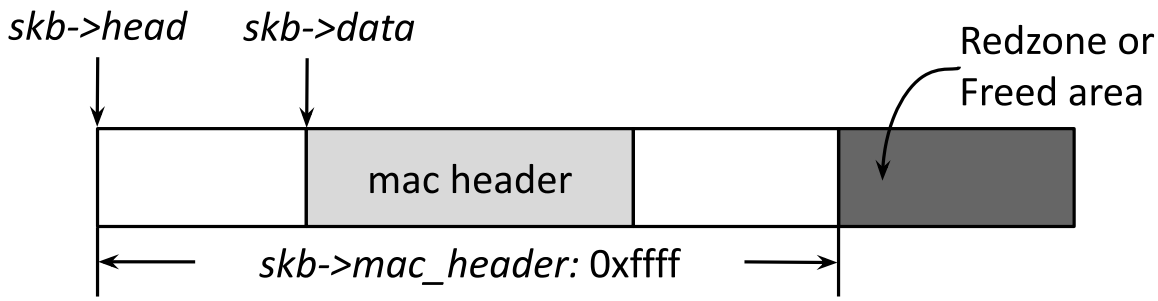

Figure 3.6: An example of demonstrating an out-of-bound bug as what is or an use-after-free error.

Factor 3: Memory Dynamics. In Linux kernel, SLAB/SLUB allocator is shared among all kernel components, responsible for dynamic memory management. Therefore, when PoC programs trigger bugs, the memory status is non-deterministic, causing the same bug to manifest different behaviors and produce duplicated reports.

Take the bug `#416dacb8` [86] as an example. Before operating an HID device, the kernel could falsely remove the device or, in other words, mistakenly deallocate the corresponding device object. When this occurs, KASAN should have reported a use-after-free error. However, due to the memory dynamics, the kernel might recycle the `slot` previously holding the device object. In this situation, rather than reporting use-after-free, the KASAN would report a slab-out-of-bound error because the accessed recycled memory region has already been set as a red zone.

Unlike the example above, which confuses KASAN to treat a freed memory region as a recycle one, another example is the bug `#96cc4b69` [87], which indicates another situation of confusing KASAN. As is illustrated in Figure 3.6, when transmitting a macvlan packet, the kernel fails to reset the `mac_header`, leaving `skb->mac_header` with the value of 0xffff. Later, when accessing the packet's data, the kernel uses the leftover value as an offset, referencing an out-of-bound memory region far away from the spot that the `skb->data` references. Due to memory dynamics, the out-

```
1 unsigned int tipc_poll(... sock) {
2   switch (sk->sk_state) {
3   case TIPC_OPEN:
4   // upstream a8750ddc
5   if (!grp || tipc_group_size(grp))
6   // net-next 594831a8
7   if (!grp || tipc_group_is_open(grp))
8   }
9 }
```

Table 3.6: The example indicating different kernel versions and branches affect the error manifestation.

of-bound access could touch either a red zone of an object or a freed spot. This causes the bug to demonstrate either an out-of-bound or a use-after-free error and eventually contribute to duplicated bug reports.

As we show in Table 3.3, among all the kernel bugs we inspected, we find that 16.5% of the bug groups are affected by this memory dynamics factor.

Factor 4: Kernel Versions and Branches. One bug can reside in several kernel versions and repository branches. When the fuzzing tools trigger the same bug in different kernel versions and repository branches, both the fuzzing programs and the crash behaviors can be different, leading to different bug reports. As shown in Table 3.6, bug `#60c25306` is a use-after-free bug existing in many kernel versions. In the commit `a8750ddc` of upstream repository, the error site is reported in `tipc_group_size`; However, in the commit `594831a8` of net-next repository, the error site is in `tipc_group_is_open`. Both are in the code block that gets executed when the condition `sk->sk_state == TIPC_OPEN` holds. The only difference is the kernel version and branch. As is depicted in Table 3.3, this factor has caused duplicated reports for 12.8% of the bug groups we analyzed.

Factor 5: Sanitizers. When performing fuzz testing against the Linux kernel, security analysts may enable or disable a sanitizer or use different sanitizers. These sanitizers are designed to capture errors in different types. Therefore, the sanitizer setup variation also contributes to the difference

```
1  void userfaultfd_event_wait_completion(...) {
2    struct userfaultfd_ctx *new;
3    new = (struct userfaultfd_ctx *)
4      ewq->msg.arg.reserved.reserved1;
5    userfaultfd_ctx_put(new);
6  }
7
8  int handle_userfault(...) {
9    struct userfaultfd_ctx *ctx =
10     vmf->vma->vm_userfaultfd_ctx.ctx;
11   BUG_ON(ctx->mm != mm);
12   if (!atomic_inc_not_zero(&ctx->refcount))
13     BUG(); // BUG if KASAN is disabled
14 }
```

Table 3.7: An example showing that switching on and off KASAN could result in different error reporting results.

in crashing behaviors and bug report duplication.

Take, for example, the bug `#0cbb4b4f` shown in Table 3.7. When forking a new process, the kernel will start an `UFFD_EVENT_FORK` event to duplicate the userfault file descriptor. If this event fails, the kernel frees the `userfaultfd_ctx` object referenced by the pointer `new` but forgets to nullify the alias pointer `vma->vm_userfaultfd_ctx.ctx` (Line 10). Later on, the function `handle_userfault` ↪ is called to handle page fault in the new process. The dangling pointer `ctx` is dereferenced to access the freed `userfaultfd_ctx` object. If `KASAN` is enabled, in Line 11, a use-after-free behavior is reported. Otherwise,the kernel continues execution until Line 13 where a BUG is reported because the reference counter of freed `userfaultfd_ctx` object is already zero.

Different from the example above, which generates duplicated reports because of switching on and off a sanitizer, another example is the bug `#250f2da4` [88], indicating another situation that causes different error behavior reporting. As is shown in Table 3.8, the root cause of this bug is that if the first argument `fqname` contains only space or the second argument `n` is zero for the function `aa_splitn_fqname`, kernel will return early in Line 9 without initializing `ns_name` and `ns_len` (Line 10 and 11). The caller function `aa_fqlookupn_profile` then uses the uninitialized `ns_name` and `ns_len` for memory access, causing errors caught by different sanitizers. If KMSAN [55] is enabled, using

the shadow memory designed specifically for uninitalization bugs, it catches an uninit-value error immediately in Line 3 and reports the root cause of the bug. However, if KASAN [53] is enabled, the kernel will not be aware of this error but propagate the uninitialized value to its consecutive execution until the error is amplified as an out-of-bounds read.

Similar to the impact of the aforementioned two factors – memory dynamics and kernel version issues, we find that 11.9% of the bug groups (with duplication) are affected by this factor (see Table 3.3).

Factor 6: Inline Function. The ways to compile the kernel code can also affect the behaviors of bugs. In particular, the compiler can make an opposite decision regarding whether to *inline a function*, which depends on the kernel configurations, the compiler (GCC or Clang) selection, and the compiler version. When this happens, we observe that one bug is triggered in different functions while in fact, they are the same program site. Our analysis shows that this factor has affected 7.3% of the bug groups we analyzed.

3.5.3 Case Study

Through our analysis, we observed interesting cases where developers were misled to develop *incorrect* patches due to their incomplete view of the diverse bug behaviors (as the bug reports were not correctly linked).

For example, bug `#416dacb8` manifests two different behaviors: `KASAN: slab-out-of-bounds Read in hidraw_ioctl` [89] and `KMSAN: use-after-free in hidraw_ioctl` [90]. The first out-of-bound read behavior was disclosed earlier. The kernel maintainers mistakenly thought that the root cause of this bug was the incorrect output range. Therefore, they developed a patch which limited the output size of `copy_to_user`. However, this bug is actually a use-after-free bug, and it has manifested out-of-bound read due to Factor-3. More specifically, the freed `slot` is recycled to hold another

```
1  struct aa_profile *aa_fqlookupn_profile(
2     ..., char *fqname, size_t n) {
3     name = aa_splitn_fqname(fqname, n, ...);
4  }
5
6  char *aa_splitn_fqname(char *fqname, size_t n, ...) {
7     char *name = skipn_spaces(fqname, n);
8     if (!name)
9        return NULL;
10    *ns_name = NULL;
11    *ns_len = 0;
12 }
```

Table 3.8: An example demonstrating the influence of different sanitizers upon bug reporting results.

object. The new object is smaller than the freed object and thus the accessed region becomes a red zone, misleading KASAN to report an out-of-bound read behavior. This incorrect patch has been committed (deployed) without being noticed. At a later time, the developers realized that the two behaviors were actually associated with the same bug [91], and the initial root cause was invalid. This example illustrates that the limited visibility to diverse bug behaviors can mislead bug patching.

3.6 Bug Report Deduplication

In this section, we provide a set of new strategies to deduplicate bug reports and prototype these strategies as a tool. Then, we evaluate this prototype tool and examine its applicability to real-world open bugs. It should be noted that our prototype is a straightforward implementation of the proposed strategies. We do not claim our prototype could pinpoint all duplicated reports. Instead, we use these straightforward implementations to answer the following three questions. ❶ What kinds of duplicated reports could be effectively and accurately pinpointed by merely following our strategies and corresponding implementations? ❷ Compared with commonly adopted stack similarity algorithms, does our prototype provide a more accurate detection? ❸ For what kinds of bug reports do we still need technical improvements for the deduplication? We hope the answers

to these questions could unveil the directions for future research.

3.6.1 Deduplication Strategies

Based on the observations from our manual inspection, we recommend five additional strategies (in addition to those that are already integrated into kernel fuzzing) that kernel developers (or Syzbot) may follow to deduplicate kernel bug reports.

First, given a bug report that looks different from previously seen bug reports in terms of the enclosed PoC and/or crashing stack trace, a kernel developer wants to determine whether the report is a duplicated copy. This developer can swap the PoC in their report and rerun it on the version of kernel specified in other bug reports. In this way, they can have different PoCs (extracted from different reports) run on the same kernel versions and thus eliminate their influence upon bug report deviation.

Second, a kernel developer can stabilize the memory layout and run the PoC programs under a relatively stable memory layout. With this, they can expect the violated memory access always lies in the same memory regions (e.g., allocated and freed memory spots) and thus minimize the influence of memory layout dynamics on bug reporting results.

Third, a kernel developer should mutate the thread interleaving and run the corresponding PoC multiple times under different thread interleaving. If the thread interleaving matters for bug report difference, this could allow a kernel developer to vary a kernel bug's panic behaviors, expose all its possible reports and thus treat these reports as duplicated bug reports accordingly.

Fourth, recall that a bug report also includes the kernel setup and configuration for the kernel fuzzing (e.g., the sanitizers they enabled). A kernel developer, therefore, should also replace the kernel fuzzing setup with the setup or configuration specified in other reports (e.g., disabling KASAN and enabling KMSAN specified in another report). By doing so, kernel developers could

have the PoC run on the same setup and thus eliminate the impact of sanitizers upon bug report difference.

Last but not least, a kernel developer can also compare the PoC program and/or the stack trace with those extracted from other reports. If the bug report duplication does not result from other factors but the difference in PoC or the sites where the kernel faults are injected, the difference between PoCs is generally less significant when the bug reports reference the same kernel bug.

3.6.2 Deduplication Tool

Following the strategies mentioned above, we propose a unified tool to facilitate bug report deduplication. The tool takes a pair of kernel bug reports as input, and passes them to five distinct technical components. For each component, it determines whether the pair of reports are duplicated because of the corresponding factor. We present each of the technical components below.

Different Sanitizers. Given a pair of bug reports, the first component examines whether the reports are duplicated because of the utilization of different sanitizers (e.g., KASAN and KMSAN). To do so, we first examine the sanitizers involved in each report. If there is only one report that indicates the utilization of a sanitizer (e.g., KASAN or KMSAN) during the kernel fuzzing, our configuration simply disables the sanitizer on the corresponding kernel specified in the report. Then, we re-run the corresponding PoC attached in that report and observe whether the sanitizer-disabled kernel still experiences unexpected termination while it takes as input the PoC program. If an unexpected kernel panic still occurs and the kernel panic is as same as the one observed on the other kernel report (i.e., the same crashing trace), we argue the report difference is contributed by the enabled sanitizer. Thus, we conclude the two reports reference the same kernel bug.

If both reports indicate the usage of a sanitizer but the sanitizers-in-use are different (*e.g.,* one uses KASAN and the other uses KMSAN), we perform the following operation. First, we take

one report as our reference and configure the kernel based on the information in this reference report. Meanwhile, we disable the corresponding sanitizer of the reference report and enable the sanitizer specified in the other report. After this configuration and setup, we re-run the PoC program attached to the reference report on the reference kernel and inspect whether the kernel panic still manifests. If unexpected kernel panics still exist and the observed panic is as same as the one demonstrated in the report, we conclude both reports link to the same kernel bug. It is simply because, after having both kernels enable the same sanitizer, we eliminate the influence of "sanitizer difference". If the unexpected panic becomes identical, the two reports are just two different exhibitions of the same bug.

Note that when we switch the reference kernel's sanitizer to the one specified in the other report, it is possible that the new sanitizer is not compatible with the version of the kernel specified in the reference report. For example, KMSAN was introduced only after kernel version 4.4, and it does not support an earlier version of the reference kernel. Moreover, KMSAN is maintained in another GitHub repository. To address this issue, we take an alternative approach that disables sanitizers on both kernels and re-runs the corresponding PoC on both sanitizer-disabled kernels. If the unexpected kernel panic still occurs and the crashing reports indicate the same termination behaviors, we conclude both reports reference the same kernel bug.

Memory Dynamics. To offset the influence of memory layout dynamics upon bug report duplication, we not only enhance an existing memory quarantine mechanism but also replace the slab allocator with the slub allocator. Under these two changes, we can minimize the memory layout dynamics. Further, by re-running the PoC programs from different reports, if the new reporting results are the same, we can safely conclude the bug reports are duplicated.

Linux community has implemented a memory quarantine mechanism [92] in both SLAB and SLUB allocators. Its basic idea is to put freed objects in a separate queue and delay their reallo-

cation. Using this mechanism, when a dangling pointer touches a memory region, the kernel can ensure that memory will have a lower chance of being recycled too quickly. As such, the quarantine mechanism could prevent a use-after-free bug exhibiting an out-of-bound activity and thus minimize the impact of memory layout dynamics. However, our manual analysis discovers, even if many Linux kernel distros have adopted this mechanism, memory dynamics still greatly contribute to the report duplication issue.

In this work, we take a closer look at the reason behind the quarantine's incompetency, and discover that the problems roots in the insufficient space of the separate queue. When the extracted PoC program performs race conditions, it usually allocates and deallocates many kernel objects. These freed objects could quickly push the quarantined memory space out of the queue and put them back into recycling. As such, the same bug could be reported differently, resulting in report duplication (e.g., sometimes reported as use-after-free and sometimes as out-of-bound access). To address this problem, we double the size of the quarantine queue and insert sleep operations after each race iteration. In this way, we not only leave sufficient space for freed objects but also prevent the race from exhausting the queue too quickly.

In addition to improving the memory quarantine mechanism, we also replace the slab allocator with the slub allocator. Based on our observations in Section 3.5, we note that both the size of the redzone and that of the kernel cache contribute to the difference in reporting results. Given slab allocators, the KASAN introduces only a relatively small memory space for the redzone. Therefore, some out-of-bound kernel bugs could jump over the redzone region, touch non-deterministic memory regions, and thus report the post-triggering activities differently. In addition, when the slab allocator is in use, out-of-bounds memory access could more easily cross the `cache` boundary, touching a `cache` totally irrelevant to the bug, and thus come up with different reporting results.

By replacing the slab allocator with the slub allocator, we can minimize the impact of redzone

and the cache upon reporting difference. This is because the size of redzone for each `slot` in SLUB allocator is larger than that in SLAB allocator. Following this setup, out-of-bounds memory access is more likely to fall in the redzone. Another reason is SLAB allocator starts allocation from the end of the pages whereas SLUB allocator prioritizes the allocation of `slots` at the beginning of the pages. With this setup, the cross-boundary situation to some extent could be mitigated. Note that in order to further mitigate the cross-boundary issue, we also increase the number of pages assigned to each `cache`. In this setup, the possibility of cross-cache-boundary access could be further reduced.

Thread Interleaving. Our idea is to mutate the thread interleaving to determine whether a pair of bug reports is duplicated. To be specific, we first analyze the bug report and extract the crashing stack traces. For the bug report tied to OOB/UAF caught by sanitizers, *i.e.,* KASAN, we also extract stack traces that allocates and frees the object that leads to kernel panic.

With these stack traces, we could know all the kernel functions that have been invoked but not returned at the time of kernel panic. Also, we can know the kernel objects that have been allocated or freed. According to a recent study [93], the functions most close to the crashing site are more likely to be the buggy function. In addition, prior research [94] also finds the crashing stack trace sometimes indicates the execution path of one of concurrent threads. As a result, we follow the work proposed in [95], focusing our consecutive analysis on the last five functions in the stack trace.

Our method could extract three different stack traces (the crashing stack trace, the stack trace related to the object free operation, and the stack trace related to the object allocation operation). For each stack trace, we select their functions most close to the crashing site. Then, we leverage SystemTap[96] to insert time delay to the starting and returning sites of these functions. Following this setup, we further extract the PoC programs from the reports under our examination, and re-run these programs multiple times (10 times by default). If we observe a crash behavior that has

already been presented in the other report, we conclude that the corresponding reports point to the same bug. It should be noted that we choose the delay time by following the heuristics applied in KCSAN (i.e., randomly selecting a value between [1,80] μs for tasks, [1,20] μs for interrupts).

Inline Function and Kernel Versions/Branches. We propose two straightforward approaches to determine whether report duplication is caused by the function inline mechanism or different kernel versions/branches.

To address the function inline issue, we simply extract and compare the dead functions from the crashing stack traces. If both crashing stacks share the same dead functions in the same sequence, then the difference between crash titles is due to an inline function, and we can conclude the two reports are about the same bug. The reason behind this design is again based on results from Section 3.2. Syzkaller uses the last crashing function to name the title of the bug report. In the process of the last crashing function extraction, Syzkaller not only skips some generic functions, but also ignores the inline functions. Take the crashing trace shown in Figure 3.7 as an example. The crashing stack traces show different function inline, and the reports deem crashing function as `nr_insert_socket` and `nr_rx_frame` respectively. In this work, by using the entire crashing stack trace, we can restore the footprint of the inline function, enable a more accurate crashing function comparison, and thus eliminate the influence of function inline for bug report deduplication.

To eliminate the influence of different kernel versions/branches, we first examine both reports and ensure the kernel information specified is truly different. With the confirmation, we then configure the kernels based on the specified information and rerun the corresponding PoC across kernels (i.e., running the PoC extracted from one report on the version of kernel specified in the other report). If we observe the swapped PoC demonstrates the kernel panic as same as that indicated in its original report, we safely conclude the pair of reports are about the same bug. The

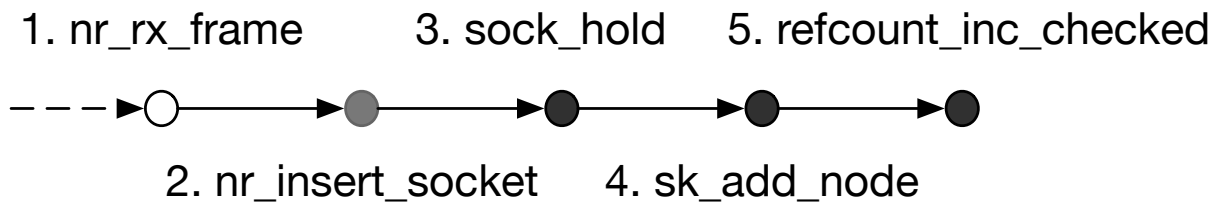

Figure 3.7: The last five functions in the crashing trace from the bug group #4638faac. ○ represents function not inlined and ● are for those inlined in both bug reports. ◍ stands for the function inline in one report but not inline in another.

rationale behind this approach is that, if both reports trigger the same bug but exhibit different behaviors only because of the different versions of the underlying kernel, swapping PoC could remove the influence of this factor and thus have the corresponding kernel demonstrate the same panic behavior under two different PoCs.

Input Difference. Given a pair of kernel bug reports, to determine whether they are duplicated by input difference, we compare the similarity of PoC programs extracted from both reports. We deem the reports are duplicated if their PoC programs are highly similar because when the PoC programs trigger the same bug, they generally use similar types of system calls and arguments.

To compare a pair of PoC programs, we first categorize system calls by using the specification provided by Syzkaller. For example, as described in the system call templates of Syzkaller, the system call `lsetxattr` and `fsetxattr` belong to the same category, taking the responsibility of setting an extended attribute for a file. With all the system calls categorized, we extract the system call sequences from each of the PoC programs and compare the sequence as follows.

First, we assign a unique ID for each category of system calls. Second, for both system call sequences extracted from the PoC programs, we map the name of the system call to the corresponding ID (the system calls in the same group share the same ID) and convert the system call sequence into a list of IDs. The arguments of each system call are the elements associated with

Method	TP Rate	FP Rate
Baseline	364/717 = 50.8%	829/79,083 = 1.05%
Our method	572/717 = 79.8%	10/79,083 = 0.01%

Table 3.9: Detecting duplicated bug report pairs.

the ID. Finally, we compute the similarity of two PoC programs by measuring the similarity of the two corresponding ID lists. In this work, we perform the list similarity measurement by using a customized version of Levenshtein distance. Due to the space limit, we present the customized distance measure in Appendix-B. We deem the reports with a similarity score greater than 0.65 as the duplicated ones. Details of the threshold selection method are also presented in Appendix-B.

Summary of Proposed Techniques. We propose five technical components to address the 6 factors of bug duplication ("Inline Function" and "Kernel Versions/Branches" are addressed together in one component). It should be noted that four of the five components are false-positive-free (the only exception is the one for "input difference"). This is because these four components determine bug duplication by explicitly resolving the differences in the original pair of reports. The technique for "input difference" is based on PoC similarity, which could have false positives. Further discussion and evaluation are in the next Section 3.6.3.

3.6.3 Ground-truth Evaluation

Dataset and Metrics. We construct a dataset to evaluate our method. First, for the duplicated reports, we directly use the final ground-truth set in Table 3.2 which contains 104 bug groups and 327 bug reports (i.e., bug titles). Then we introduce another 73 bug reports randomly selected from the non-duplicated bug titles (we manually confirmed the non-duplication). In total, the dataset contains 400 bug reports covering 177 unique kernel bugs (i.e., 177 bug groups).

Considering our method takes *pairs of bug reports* as inputs, we format the dataset by exhaus-

Factor	GT Pairs	Detected Pairs	Contributed FP
Different Sanitizers	33	33	0
Memory Dynamics	178	178	0
Inline Function	43	43	0
Thread Interleaving	122	62	0
Input Difference	341	256	10
Total	717	572	10

Table 3.10: Performance breakdown for each factor and its sub-method. "GT Pairs" means the number of ground-truth duplicated pairs associated with each factor. "Detected Pairs" means the number of duplicated pairs detected by each factor-specific method.

tively pairing the reports. This generates a ground-truth dataset of 717 duplicated pairs (positive) and 79,083 non-duplicated pairs (negative).

We consider two common evaluation metrics. *TP (True Positive) rate* is the ratio of the real duplicated pairs that are successfully detected (marked out) by our method. *FP (False Positive) rate* is the ratio of the non-duplicated pairs that are incorrectly detected as duplicated pairs. Considering our dataset is skewed, we also report the raw numbers (true positives and false positives) along with these rates. In this section, our analysis is focused on *bug report pairs*. Further discussion of the *bug-group* level performance is in Appendix-C.

Baseline Method. It is difficult to find a direct baseline since there is little work on bug deduplication for Linux kernel. The most relevant deduplication heuristics for *kernel space* are already used by Syzkaller. Our method is built on top of Syzkaller's deduplication results (i.e., their bug titles) to make further improvements. There are indeed deduplication methods used in *user space* for fuzzing tasks, which could be used as a baseline. Here, we choose a popular stack similarity algorithm from ClusterFuzz [7] which can work well with the stack traces of Syzkaller's kernel crash reports. We follow their deduplication policy and re-implement it to compare the stack traces of kernel crash reports. We present the detail of our re-engineering effort in Appendix-D.

Detecting Duplicated Bug Pairs. The detection results are summarized in Table 3.9. We show that our method achieves good performance and outperforms the baseline. More specifically, our method detects 572 of the 717 duplicated pairs with a true positive rate near 80% (50.8% for the baseline). At the time, we have introduced 10 false positives (0.01%) which is much lower than the baseline (829 false positives, 1.05%). The result confirms that simply comparing the stack traces is insufficient to detect duplicated bugs (baseline method). To further understand the sources of errors (especially false positives), we break down duplicated reports based on the contributing factors, as shown in Table 3.10. For example, there are 33 duplicated pairs caused by "Different Sanitizers". We find that all 33 pairs are successfully marked by the proposed method with 0 (zero) false positive. Table 3.10 confirms that the only source of false positives (FP) is the method for "Input Differences". All other methods proposed for other factors are FP-free (by design). At the same time, we observe that the proposed methods for "Different Sanitizers", "Memory Dynamics", and "Inline Function" have a perfect true positive rate (100%). While the method for "Thread Interleaving" has missed some duplicated pairs, it does not introduce any false positives.

Detailed Error Analysis - FN. Our methods for "Input Differences" and "Thread Interleaving" have missed some truly duplicated pairs. We manually examine these cases and find that, under "input differences", the errors are mostly caused by the PoC programs which have a low similarity in their system call sequences and arguments. Take the duplicated reports [97] and [98] for example, one PoC invokes 16 system calls while the other involves only one system call. Further work is needed to address such cases with auxiliary information. We argue that this does not necessarily dismiss the value of the proposed technique since it still recovers 256/341 (75%) of the duplicated pairs caused by input differences.

For "thread interleaving", we find that the missed pairs are mostly caused by three reasons.

First, not all crashing stacks are directly involved in the thread synchronization. As such, inserting time delays to the functions extracted from crashing stacks does not always work. Second, for certain cases, the time delay needs to have a specific value to trigger the expected results. As such, a random time delay could be insufficient. Third, the inserted time delay could potentially influence the thread interleaving but does not guarantee the thread synchronization to occur as expected. Overall, we believe further work is needed to improve the time delay insertion (based on static and dynamic program analysis) and perform more fine-grained thread scheduling control for deduplication.

Detailed Error Analysis - FP. As shown in Table 3.10, the "input difference" method is the only source of the 10 false positives. In practice, conservative developers (or Syzbot) could use all the other FP-free techniques to automatically group bug reports. For the input difference method, Syzbot can use it to make recommendations to developers on "potentially duplicated" reports or optimize the maintainer assignment process (i.e., assigning the same set of maintainers to the detected bug pairs). Because it has FP (even though small), we do not recommend using it to automatically merge/remove duplicates. We have manually examined these false positives, and determined that their PoC programs have highly similar system calls. For example, bug reports [99] and [100] are about different bugs, but the two PoCs are using the same pseudo system call `syz_usb_connect` to trigger the bug. The only difference between the PoCs is the arguments passed to the system call. Such a minor difference results in a high similarity measure and thus false positives.

As discussed in Section 3.3.2, false positive cases have a lower impact because Syzbot can effectively handle them with continuous fuzzing. If a few corner cases are grouped incorrectly by the *input difference method*, Syzbot can ensure that these bugs will not be ignored by kernel developers. More specifically, after a patch is developed, Syzbot will continue to test the patched kernel version. If not all bugs are fixed, Syzbot will continue to file crash reports, and kernel

developers will work on these reports to patch the remaining bugs.

Applicability. While this paper is focused on Linux kernel, the proposed techniques are applicable to other open-source kernels such as FreeBSD, NetBSD, and OpenBSD. Note that *Syzbot* already supports fuzzing these kernels, and it has already generated crash reports for them.

Scalability. We want to briefly discuss given the fact that deduplication requires testing a large number of bug pairs. During our analysis, we find that the runtime for analyzing each bug pair varies significantly. There are two reasons. First, if one strategy can successfully confirm duplication, we will stop testing the remaining strategies. Second, we only apply a specific deduplication strategy when their corresponding conditions are met. The applicable strategies for each pair may differ, which affects the runtime. For example, if a pair of bug reports have different sanitizers, we will apply the strategy for *different sanitizers*. In this case, the most time-consuming part is kernel recompilation. It only takes about 1 minute to run PoC testing in the QEMU VM. However, it takes an 8-core desktop 15–20 minutes (depending on the kernel configuration) to recompile the Linux kernel.

When applying these deduplication techniques in practice, we do not need to compare a new bug report (title) with all the historical reports. Instead, we only need to compare it with "Open Bugs". The open bug queue usually contains $< 1,000$ open bugs. Given the rate of bug discovery (e.g., about ten bugs per week), this overhead is manageable. Furthermore, we can run deduplication strategies in parallel (*e.g.,* compiling Linux kernel with different configurations or run the same PoC in different VMs) to improve the runtime efficiency. As a reference point, it took about two weeks to run our ground-truth evaluation, which involves testing about 80,000 bug pairs using two commodity servers. With improved parallelization and additional computation resources, we argue such overhead is acceptable to companies such as Google.

3.6.4 Applying Our Methods to Open Bugs

The above ground-truth experiments confirmed the effectiveness of our methods. We next apply our methods to real-world *open bugs* to catch previously unknown duplication.

Analyzing Open Bugs. In January 2021, we collected all the bug titles from the "Open bugs" queue on the Syzbot dashboard. These bugs are have been reported but are not yet fixed. We obtained 652 bug reports that contained reproducers at the time of the experiment. While we can exhaustively apply our automated techniques to all the bug report pairs (212,226 pairs), to validate the correctness of results, we need to select a small subset of pairs for the time-consuming manual testing and validation.

We first extracted the crash function, crash type, the PoC program, and the stack trace from each bug report. Then, for each bug report pair, we automatically computed the similarity of their crash description, the PoC program using the method discussed in Section 3.6.2. We kept the bug report pair for further examination if it satisfies one of the following criteria: ❶ The two crashes occur in the same function; ❷ The two bug reports manifest the same crash type associated with the same subsystem (i.e., the crash functions are defined in the same directory); ❸ The similarity of the PoC programs in the two bug reports is greater than 0.65; ❹ The five recently called functions in the two stack traces are the same. After the filtering, we have 455 pairs left for further investigation.

Findings. We confirmed that our techniques were practically effective in identifying *previously unknown* bug duplication. In total, we identified 27 groups of duplicate bugs. These bug groups involved 66 bug titles and 66,594 crash reports. By examining the causes of duplication, we find that the most commonly observed factors are 'inline function" (affecting 13 bug groups) and "different inputs" (affecting 8 bug groups). For the rest of the factors, each affects ≤ 7 bug groups.

Since we do not have the "ground truth" for open bugs, we randomly select 20 groups of du-

plicate bugs (flagged by our techniques) and 20 non-duplicate groups to validate the effectiveness of our methods. Through manual analysis, we do not observe any false positives. We only find 1 false negative from the non-duplicate groups. The underlying reason is that the PoCs between the two groups are highly different, with different syscall sequences and different syscall arguments. Our tool cannot effectively group them together.

When analyzing the flagged duplicate bug groups, we have several key observations. First, for cases that are affected by "inline function", our tools can effectively identify the functions with different inline status in different reports. For example, in the bug group of `WARNING: refcount bug in qrtr_node_lookup` and `WARNING: refcount bug in qrtr_recvmsg`, our tool can identify the function `qrtr_node_lookup`, which is inline in one report and non-inline in the other report. Second, there is a case identified by "kernel versions and branches". We find two similar-looking but actually different key functions between two different kernel versions: `sctp_ulpevent_notify_peer_addr_change` and `sctp_ulpevent_nofity_peer_addr_change`. Note that one function name contains the keyword `notify`, the other function name contains the keyword `nofity` (notify vs. nofity). Third, there is a case detected by "different sanitizers", where we switch from KMSAN to KASAN in one crash report and recompile the source code. After that, we observe the same crash behaviors in another crash report. Fourth, for a case affected by "memory dynamics", our method helps to stably manifest the crash behavior in another report. Finally, our PoC similarity analysis has helped to verify cases affected by "input difference". For example, the PoCs for `general protection fault in l2cap_sock_getsockopt` and `general protection fault in sco_sock_getsockopt` have a high similarity except for the socket type in `syz_init_net_socket`.

When analyzing open bugs, we again find a similar case to that described in Section 3.5.3. This bug group contains one bug report that shows a `UBSAN: shift-out-of-bounds in mceusb_dev_recv` behavior [101] and another report shows a `UBSAN: shift-out-of-bounds in mceusb_dev_printdata`

behavior [102]. In fact, there has already been a patch developed and discussed in the community for the first behavior. This patch adds a sanity check only in the function `mceusb_dev_recv`. However, after analyzing the root cause, we concluded that the two bug reports are indeed duplicated and more importantly, the patch only mitigates the first behavior. This again confirms that a limited view of the diverse bug behaviors indeed leads to incomplete patches.

Result Sharing and Communication. A recent work [50] shows that knowing the multiple behaviors of the same bug can help to improve crash analysis. To this end, we have further reported the duplicated bugs we found to the Syzbot dashboard and notified the corresponding developers. We hope our findings can help with ongoing bug analysis.

At the same time, we have communicated our results and findings with the Syzkaller and Syzbot team. The initial response from the team was positive, echoing the importance of finding the duplicated bugs given the backlog of open bugs they are currently accumulating. We are in the process of introducing our tool, and exploring the opportunity to integrate it into Syzbot to improve the current deduplication system.

3.7 Related Work

In this work, we perform a large-scale measurement to study the bug report duplication phenomenon on Syzbot, with the purpose of improving root cause diagnosis and facilitating bug patching. As such, we consider the works in the following directions as related.

Bug Report Deduplication. Prior works have explored different methods to deduplicate bugs discovered by automatic tools. One method is to leverage stack trace in the bug report to determine whether two reports are referring to the same bug. Fuzzing tools, including SmartFuzz [95], VUzzer [103], FuzzSim [104], CERT BFF [105], and Syzkaller **Syzkaller**, generate

a stack hash using the function name, line number, and file name on the call stack for deduplication. Other works (*e.g.,* [106], [107]) calculate stack edit distance and TF-IDF for deduplication. ReBucket [108] measures the similarities of call stacking using an Dependent Model (PDM) algorithm. Another method considers the basic block transition in the execution path, which is widely employed by AFL-based fuzzers (*e.g.,* [47], [50], [109], [110], [111], [112], [113], [114]) Unfortunately, a study by Klees *et al.* [115] shows that this approach could yield excessive false positives and false negatives. Finally, Tonder *et al.* [116] propose a method that groups the bugs with similar fixes. Despite these efforts, none of the previous works are focused on reasoning the factors that have caused different crash behaviors and designing solutions to handle these factors (the main focuses of our work).

Empirical Studies of Kernel Bugs. Researchers have performed empirical studies of kernel bugs, with different purposes compared to our paper. For example, Abal *et al.* [77] have studied 42 bugs in the Linux kernel. They observed that variability bugs do not exclusively belong to any particular bug types, error-prone features, or source code locations, while the variability property has greatly increased the complexity of bugs in the Linux kernel. PDiff [78] performed a comprehensive study to understand the patch presence testing problem. They identified two essential challenges in the testing: third-party code customization and diversities in the building configuration. Xu *et al.* [81] empirically studied real-world Android kernel vulnerability patches. They found that the code changes of security patches are generally small compared to non-security patches and large security patches usually contain several small individual patches. Li and Paxson [117] conducted an empirical study of security patches to understand their development life cycles. They show that security patches are more localized (than other non-security patches) but usually suffer from a long delay. Mu *et al.* [60] analyzed crowd-reported vulnerability reports to assess their reproducibility. Our work is the first to study the factors causing the report duplication problems for kernel bugs. In

addition to the empirical study, we also provide suggestions to deduplicate bug reports to facilitate root cause diagnosis and bug patching.

3.8 Conclusion and Future Work

We discover the duplicated kernel bug reports are prevalent and could potentially cause the delay of kernel bug remedy. Through intensive manual efforts, we analyze the root cause behind the duplicated reports and summarize six key factors directly contributing to report duplication. Under the guidance of our discovery, we prototype a series of deduplication strategies. We conclude the newly proposed deduplication strategies could group a majority of duplicated kernel bug reports correctly and even facilitate correct kernel patch development. As is described in Section 3.6.3, our proposed deduplication method is merely the initial exploration. Our future research will focus on exploring more advanced technical approaches to better cluster duplicated reports or in other words further reduce the errors introducing in the report grouping. Besides, we will study how to use the grouped reports to better diagnose the root cause of corresponding kernel bugs and thus further benefit bug remedy. Finally, we will also study the bug duplication problem for other OSes (e.g., Windows and XNU).

CHAPTER 4
MITIGATING SECURITY RISKS IN LINUX WITH KLAUS – A METHOD FOR EVALUATING PATCH CORRECTNESS

4.1 Introduction

In this section, we focus on the problem of ensuring the correctness of Linux kernel patches. The Linux kernel is an extraordinarily large and evolving codebase, and its complexity makes it highly susceptible to bugs. Automated bug-finding systems—such as syzkaller and related kernel testing frameworks [5], [118], [119], [120], [121], [122]—continue to uncover a substantial number of issues, underscoring the need for constant maintenance and rapid patching by the kernel community. Because the Linux kernel forms the foundation of countless computing platforms and security-critical systems, patch development is not merely routine maintenance: it is a central responsibility that directly impacts system reliability and security. Motivated by the importance of producing correct and robust kernel patches, this section presents our work on analyzing, evaluating, and improving the correctness of real-world Linux kernel patches.

To tackle the large volume of kernel bugs, the Linux community relies on crowd-sourcing to remediate them. Over the past ten years, thousands of bugs have been analyzed and fixed by numerous kernel researchers. Despite these efforts, many kernel patches prove to be challenging to develop correctly on the first try. From 2012 to 2022, over 3,500 patches that aimed to address one issue actually introduced new bugs. Unfortunately, some of these new bugs were even more exploitable than the original bugs they were trying to fix, as demonstrated in Section 4.2.2 and 4.7.4.

Our manual analysis of hundreds of incorrect kernel patches reveals that the existence of these

incorrect patches can be attributed to two main factors. First, the complexity of the Linux kernel makes it difficult for researchers and developers to ensure their patches do not disrupt the existing kernel logic. Second, there is a lack of effective mechanisms for assessing patch quality. Today, the only way to evaluate a patch is through regression testing, where the patch is applied to the kernel and its ability to eliminate the original bug's error behavior is tested by re-running the trigger input. However, as demonstrated in the following sections, the absence of an error behavior does not guarantee the patch's correctness.

One solution to address this pressing issue is to use widely adopted kernel fuzzing techniques such as Syzkaller [5], SyzVegas [123], StateFuzz [124], and HFL [125] to test the patched kernel. However, using these methods may result in inefficiencies as they often prioritize higher kernel code coverage, leading to a waste of time and resources on code that has no relation to the newly introduced bug.

In this work, we propose a new technical method to assess kernel patches, called `KLAUS` (standing for KerneL pAtch qUality aSsessor). Unlike traditional kernel fuzzing techniques, `KLAUS` first conducts a static analysis of the code before and after the patch is applied. It then uses abstract interpretation to track changes in read and write operations made by the patch. Our manual analysis of 182 incorrect Linux kernel patches, described in Section 4.3, shows that newly introduced bugs in incorrect patches primarily result from changes in read and write operations. Based on this observation, our static analysis informs a directed fuzzing mechanism that uses changes in read and write operations as indicators to guide the fuzzer toward code and context relevant to the patch.

Different from many existing directed fuzzing schemes (e.g., SemFuzz [126] and AFLGo [127]), our method does not primarily focus on optimizing input to reach specific program sites. Going beyond reachability, it also takes into account the order and type of patch-altered read and write operations during kernel fuzzing. Additionally, our method utilizes branch-resolving mechanisms

```
1  // description
2  nfc: fix refcount leak in llcp_sock_connect()
3  Fixes: c7aa12252f51 (NFC: Take a reference ...)
4  // diff file
5  diff --git a/net/nfc/llcp_sock.c b/net/nfc/llcp_sock.c
6  --- a/net/nfc/llcp_sock.c
7  +++ b/net/nfc/llcp_sock.c
8  @@ -704,6 +704,7 @@ static int llcp_sock_connect()
9  ...
10 + nfc_llcp_local_put(llcp_sock->local);
11 }
```

Listing 4.1: The snippet of commit `8a4cd82d62b5` that includes a description and a diff file.

to further improve KLAUS's ability to detect incorrect patches. Our evaluation shows that KLAUS outperforms Syzkaller in detecting incorrect patches. With KLAUS, we discovered 30 incorrect patches from hundreds of Linux kernel patches. So far, 25 false patches have been confirmed and fixed. Out of all the bugs we discovered, we also found 3 that offered attackers higher exploitability than the original bugs. We successfully demonstrated privilege escalation against the latest Ubuntu and Android systems using these 3 vulnerabilities and received $90,337 bug bounty rewards from Pwn2Own[128] and Google KCTF[129].

This work, to the best of our knowledge, is the first of its kind to explore the correctness of Linux kernel patches. It highlights a concerning reality that an improperly developed patch can result in a more severe security vulnerability. KLAUS, as a new testing tool, offers a more effective and efficient method for uncovering bug patch incorrectness. Moreover, it empowers kernel developers to evaluate the quality of their patches, potentially reducing the risk of transforming a non-exploitable bug into a highly exploitable vulnerability.

In summary, this paper makes the following contributions.

- We have conducted a thorough examination of 182 incorrect Linux kernel patches and uncovered crucial insights about the root causes of incorrect patches. Our findings highlight the limitations of current methods for identifying incorrect patches and inspired the development of KLAUS.

- We propose a new method that leverages abstract interpretation to capture the change of the variables' read and write operations made by a kernel patch. Using the extracted read-and-write operation alteration, we also propose a new directed fuzzing mechanism for incorrect patch identification.

- We implemented KLAUS and thoroughly evaluated its performance by using hundreds of real-world Linux kernel patches. We have uncovered and reported 30 incorrectly developed and/or deployed kernel patches, resulting in many of these patches being immediately re-patched. Upon acceptance of this submission, we plan to make KLAUS available to the community by open-sourcing it.

The rest of this paper is organized as follows. Section 4.2 introduces the background and example of incorrect kernel patches. Section 4.3 presents our key observation from incorrectly developed kernel patches and describes the high-level idea of KLAUS. Section 4.4 and 4.5 present the technical details of KLAUS. Section 4.6 discusses our implementation details. Section 4.7 evaluates the effectiveness of KLAUS on real-world Linux kernel patches. Section 4.8 provides the discussion of related work, followed by the conclusion of the work in Section 4.9.

4.2 Background & Working Example

This section delves into the background of Linux kernel patches. To clarify the issue of improperly created patches, we also provide a real-world example of how a non-critical kernel bug can turn into a vulnerable Linux kernel exploit.

```
1  struct nfc_llcp_sock {
2     struct nfc_llcp_local *local; // NFC device
3  };
4
5  void *nfc_llcp_local_get(struct nfc_llcp_local *local) {
6     kref_get(&local->ref); // local->ref++
7  }
8
9  int nfc_llcp_local_put(struct nfc_llcp_local *local) {
10    if (local == NULL)
11       return 0;
12    // local->ref-- and free local via
13    // local_release() if local->ref == 0
14    return kref_put(&local->ref, local_release);
15 }
16
17 static int llcp_sock_bind(struct nfc_llcp_sock *llcp_sock) {
18    nfc_llcp_local_get(llcp_sock->local);
19    if (llcp_sock->ssap == LLCP_SAP_MAX) {
20       // 1st patch which is incorrect
21       nfc_llcp_local_put(llcp_sock->local);
22       llcp_sock->sock = NULL; // 2nd patch
23       goto put_dev;
24    }
25 put_dev:
26    return -EADDRINUSE;
27 }
28
29 void nfc_llcp_sock_free(struct nfc_llcp_sock *sock) {
30    if (sock->local != NULL)
31       nfc_llcp_local_put(sock->local);
32 }
```

Listing 4.2: The code snippet of the Linux kernel that contains two patches. The 1st patch in commit `8a4cd82d62b5` fixes a memory leak bug but incorrectly introduces a double free vulnerability. This new vulnerability resided in the kernel for almost two months until finally being fixed by the 2nd patch in commit `c61760e6940d`.

4.2.1 Kernel Commit and Patch

As shown in List 4.1, a kernel commit contains two major parts. The first part is a description starting with a title (line 2). If the commit is to patch a bug, the description will include a line with "Fixes:" as the prefix (line 3). The commit ID following this prefix is a previous commit that introducing the bug. The second part is a diff file which lists code modification to the kernel. It illustrates the location of code modification - in function `llcp_sock_connect` (line 8) of file `net/nfc/` ↪ `llcp_sock.c` (line 5-7) from kernel code line 704. The code modification is further specified in

line 10 which starts with a "+" symbol and inserts a calling to kernel function `nfc_llcp_local_put`. Correspondingly, if a line will be removed, it starts with a "-" symbol in diff file. By applying the code modification in the diff file, the vulnerable kernel is patched.

A patch typically goes through three phases before finally being merged into the upstream kernel. In the first phase, kernel developers craft a patch draft in the local workspace and post it to a corresponding mailing list for review and discussion. If other developers and maintainers reach an agreement, then this patch will be merged into the git tree of the kernel subsystem that contains the bug. Finally, when a merge window is open, all patches in the subsystem will be pushed to the upstream kernel. In the second and third phases, automatic testing will be performed to examine the patch quality. Common testing techniques include regression testing (*e.g.,* , kselftest [130], 0-day kernel test [131], and KernelCI [132]) and Syztest [133]. They replay the input PoC program that triggers the bug against patched kernel. If no kernel error is observed, they conclude the correctness of the patch. However, as we will show in the following session, these existing techniques are insufficient to detect mistakes in patches.

4.2.2 From Memory Leak to Double Free

The patch shown in List 4.1 is to fix a memory leak bug. In List 4.2, we include other relevant kernel code illustrate the root cause. As shown in List 4.2, all `nfc_llcp_sock` sockets share one `NFC` device which is represented in the `local` field (line 2). To use the device, the kernel calls the function `nfc_llcp_local_get` (line 5) to increase its reference count (refcount for short) of `nfc_llcp_sock->local` ↪ . After finishing the usage, the kernel calls the function `nfc_llcp_local_put` (line 9) to decrease the refcount for balance. When the refcount is decreased to zero, it means the `NFC` device is no longer used. Then, the kernel will call function `local_release` to free `nfc_llcp_sock->local` (line 14).

The root cause of the memory leak is in function `llcp_sock_bind`. In line 18, the refcount of

llcp_sock->local is increased to use NFC device. However, if llcp_sock->ssap is equal to LLCP_SAP_MAX (line 19), the kernel jumps to the label put_dev and returns the caller without decreasing the refcount, resulting in imbalance. As a consequence, the memory referred by llcp_sock->local is never freed and reclaimed, causing memory leak. To rebalance refcount, the patch shown in List 4.1 inserts the call to nfc_llcp_local_put (line 21) in function llcp_sock_bind.

This patch is straightforward but incorrect because it mistakenly introduces a double free vulnerability. To be specific, after the NFC device is probed, its refcount is initialized to 1. Then, the kernel creates two nfc_llcp_sock sockets. Binding the 1st socket to the device, the kernel executes line 18-21, which increases the refcount to 2 (line 18) and decreases it to 1 (line 21). The refcount is well maintained so far. However, when the 1st socket is closed, the kernel will call function nfc_llcp_sock_free (line 29) to destroy it. Function nfc_llcp_sock_free further calls nfc_llcp_local_put, decreasing the refcount to 0 and freeing the memory referenced by nfc_llcp_sock ↪ ->local. Recall that all nfc_llcp_sock sockets share one NFC device by having their local refer to the same nfc_llcp_local. After the nfc_llcp_local is freed, the local field of the 2nd socket becomes a dangling pointer. Then, binding the 2nd socket, the kernel decreases the refcount of nfc_llcp_local->local to 0 again in line 21, which frees the memory referenced by nfc_llcp_local-> ↪ local for the second time.

Compared with memory leak, double free is widely believed to be more exploitable [134]. This double free was assigned with CVE-2021-23134[135] and kernel developers crafted another patch with commit c61760e6940d [136] for remediation. This 2nd patch nullifies nfc_llcp_sock->local ↪ in line 22 so that nfc_llcp_local_put is not called in line 31 when the 1st socket is destroyed. As such, the refcount is not decreased to 0, avoiding mistaken free of the memory referenced by nfc_llcp_local->local.

In summary, the 1st patch in commit 8a4cd82d62b5 [137] tries to fix a memory leak but incor-

rectly introduces a new double free vulnerability. This new vulnerability was not detected by any testing techniques and resided in the kernel for almost two months. To fix the new vulnerability, kernel developers had to revisit the same piece of code, diagnose the root cause again, and craft the 2nd patch in commit `c61760e6940d` [136] for remediation.

4.3 Empirical Analysis and Design Overview

We collected incorrect patches and analyzed them to understand why patches can mistakenly introduce new vulnerabilities and why existing testing techniques are unable to detect them.

4.3.1 Incorrect Patch Collection

We took the following steps to collect a set of incorrect patches for empirical study. First of all, we crawled the upstream Linux kernel reports[138] to get all commits with the "Fixes" tag from 2012 to 2022. Recall that, the "Fixes" tag in a commit indicates that the purpose of this commit is to patch a bug in the kernel. To filter out incorrect patches from these commits, we tried to construct fixing relationships. More specifically, we traversed from the newest commit c_0 in this set through a depth-first search. If the commit c_1 referred to by the "Fixes" tag in c_0 is also a patching commit, we built a fixing relationship between them $c_0 \rightarrow c_1$ and continue the traversing from c_1. We repeated this process until all commits are examined. Finally, we got 4,534 chains representing fixing relationships. There are 8 longest chains built in this approach, containing 6 commits across the span of 1,115 days at most. In other words, it took 1,115 days and 5 patches for kernel developers to fix the initial bug. All commits in a chain, other than the head and the tail, are fixing the previous commit and in the meanwhile fixed by the next commit. Therefore, these commits are incorrect patches we are searching for. In total, we found 3,706 incorrect patches.

Considering the huge amount of incorrect patches, we sampled 182 (5%) from them for manual

analysis. We only sampled the reports generated by Syzkaller so that we can estimate their security impacts and reproduce them using the essential information in Syzkaller's reports.

Our manual analysis focuses on understanding why these patches are incorrect. We built a team of three security analysts. All team members have rich experience in kernel patching and vulnerability exploitation. Each incorrect patch was randomly assigned to two analysts, and conclusions were drawn only when their analysis results converged. It took the whole team about 1,080 man-hours to finish. On average, each commit requires nearly 6 hours to conclude.

4.3.2 Key Observations

Through the analysis, we gain two key observations which not only explain why existing techniques are incapable of detecting incorrect patches but also enlighten the design of KLAUS.

Observation 1: Old and new vulnerabilities share similar contexts. Intuitively, the incorrect patch fixes the old vulnerability and, at the same time, introduces a new vulnerability. Therefore, the kernel codes pertaining to the new vulnerability have a large overlap with those pertaining to the old vulnerability, which means the contexts are similar. In our working example, to trigger the memory leak, a `nfc_llcp_sock` socket must be created and bound to the `NFC` device. To misbalance the refcount, `llcp_sock->ssap` must be equal to `LLCP_SAP_MAX`. These conditions are also needed to trigger the double free vulnerability introduced by the incorrect patch.

This observation hints that to construct a PoC program (i.e., an input) to trigger the new vulnerability, we can reuse the context provided by the PoC program for the old vulnerability to save time. From this perspective, it seems regression testing is an ideal solution for incorrect patch detection because it replays the PoC program and thus the same context. Same context can reproduce the old vulnerability but to trigger the new vulnerability, the context must be varied. In our working example, on the basis of one `nfc_llcp_sock` socket used to trigger memory leak, binding a

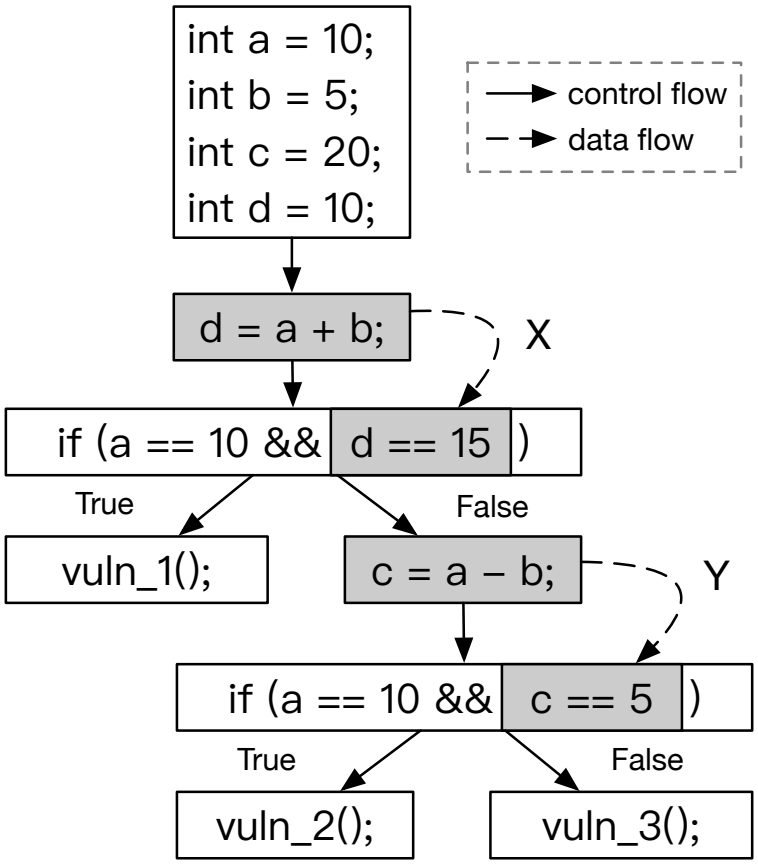

Figure 4.1: A synthetic program to explain how AWRPs introduced incorrect patches finally result in new vulnerability.

second `nfc_llcp_sock` socket to the `NFC` device is critical to manifest the double free.

Therefore, an efficient strategy to explore the huge codebase of the kernel and trigger the new vulnerability is to stick the kernel execution to the old context and in the meanwhile vary it subtly. To this end, an instinctive solution is to perform direct fuzzing towards code changes introduced by the patch (*e.g.,* , [126], [139]). Since the changes are close to the vulnerable code, fuzzing towards them is likely to preserve the context. By covering the changes introduced by the patch, the context is varied to accommodate to the new vulnerability. However, this intuitive design can hardly work in practice. Again looking at our working example, we can note that the patch does

not modify `nfc_llcp_sock_free` function. A naive directed fuzzing will not set this function as the target and make attempts to reach it. As a consequence, no matter how many times the added `nfc_llcp_local_put` function is executed, the expected double free will never be triggered.

Designing a smarter directed fuzzing to trigger the new vulnerability resulting from incorrect patch, we, therefore, need more in-depth guidance rather than merely based on code changes. A recent work PatchVerif [140] proposes a method for patch applied to robotic vehicles. It executes the code before and after modification and then monitors the variation of the vehicle's speed, location, and altitude etc. If the variation is significant, PatchVerif concludes that the corresponding code change might be faulty. While this method demonstrates effectiveness in patch assessment, it is designed specific to robotic vehicles. As a result, it is not able to be applied to our problem.

Observation 2: New vulnerability results from Altered Write-Read Pairs (AWRP). The old vulnerability is fixed because the code changes introduced by the patch make modification to the the data flow and control flow of the kernel. Due to these changes, the triggering condition of the old vulnerability can no longer be satisfied. In our working example, the patch decreases the refcount by inserting `nfc_llcp_local_put` before returning from function `llcp_sock_bind`. This new write operation of the refcount rebalances the refcount and prevents memory leak. However, from a broader perspective, this new write operation in `nfc_llcp_local_put` is unintentionally associated with the read of the refcount in `nfc_llcp_sock_free` function, which causes an unexpected double free.

Through empirical study, we discovered that it is not an coincidence that the new vulnerability is caused by a write-read pair like the one in our working example. Further, the root cause of all incorrect patches can be attributed to the occurence of Altered Write-Read Pairs regarding the same variable (AWRP for short). Generally, we consider a pair of write-read operations of the the same variable is an AWRP if one of the following situations happens. ❶ A new write or read operation

is added by the patch. This new operation can be matched with existing read or write operations to form a new pair. ❷ An existing write or read is removed by the patch. As a result, an existing pair is eliminated. ❸ The writing value or the read behavior is changed. In other words, either the interesting variable is assigned with a different value or the interesting variable is used in different way. This situation can be deemed as a combination of the first and second situation: removing an old write/read and adding a new write/read. ❹ The path condition to execute the write or the read operation in an existing pair is changed.

In Figure 4.1, we use a synthetic example to illustrate when a patch is applied, how AWRPs affect the data flow and the control flow, finally resulting in new vulnerabilities in the kernel. There are two distinct write-read pairs in Figure 4.1: `d = a + b` matches with `d == 15` with regard to variable `d` in pair X and `c = a - b` matches with `c == 5` with regard to variable `c` in pair Y. Now we consider three possible patches that can be applied individually. In our first synthetic patch, we assume that `d = a + b` is newly added and all others exist in the original kernel. According to situation ❶ mentioned above, pair X is an AWRP because `d = a + b` is a new write operation of variable `d`. With this new write, the condition in the first `if` statement (*i.e.,* , `if(a == 10 && d == 15)` ↪) will be evaluated to `True`. Then the kernel will take the branch and triggers the vulnerability `vuln_1`. In our second synthetic patch, we assume that `d = a + b` is removed and all others remain unchanged in the kernel. According to situation ❷, pair X is an AWRP because `d = a + b` is a deleted write operation of variable `d`. As a result, the code in the false branch of the first `if` ↪ statement is activated. It is no longer a piece of dead code. According to situation ❹, now pair Y is also an AWRP. After executing `c = a - b`, the kernel will evaluate the second `if` statement to be `True` and takes the branch to trigger the vulnerability `vuln_2`. In our third patch, `d = a + b` is removed and `c = a - b` is changed to `c = a + b`. According to ❸, pair Y is an AWRP. The condition in the second `if` statement will be evaluated to `False` which leads to the triggering of vulnerability

vuln_3.

The AWRP methodology provides a way to analyze the impacts of patches on data and control flow. Our findings show that incorrect patching results from mistakenly altering write-read pairs. Additionally, we have observed that incorrect patches may trigger kernel errors when the AWRP-related code is executed.

4.3.3 Design of KLAUS

Based on the key observations discussed above, we present the design of KLAUS which aims to generate concrete input to execute kernel code related to patch and thus demonstrate kernel errors if the patch is incorrectly developed or deployed.

The key idea of KLAUS is to leverage AWRP as the indicator to efficiently search the huge codebase of the Linux kernel. Through executing the write operation first and then the read operation in an AWRP, KLAUS generates the desired input sequence. Following this idea, the first component of KLAUS utilizes static analysis techniques to identify AWRPs caused by the patch. The static analysis component starts by profiling the influence of AWRPs through abstract interpretation. By collecting instructions the abstract states of which are changed because of the patch, KLAUS constructs AWRPs by pairing write instructions and read instructions that are pertaining to the same variable.

With the set of AWRPs as the guidance, the second component of KLAUS performs directed fuzzing. As mentioned above, vulnerabilities related to incorrect patches can be triggered by executing AWRPs. Inspired by this, our directed fuzzing first makes attempts to reach the write operation and then switches to reach the read operation belonging to the same pair. As we will elaborate in Section 4.5, KLAUS uses both AWRP-related code and type coverage as the feedback for exploration. Considering that write and read operations can be dominated by branches that are hard to

Input: Function F and its CFG with e as the entry node ;
Output: $S[]$;
Initialize: $waitList = \{e\}, S = \{e \rightarrow \top, \forall n \neq e \rightarrow \bot\}$;
while $\exists$ *n* $\in$ *waitList* **do**
    $waitList = waitList - \{n\}$;
    $s' = Transfer(S[n], n)$;
    foreach *n' $\in$ Successors(CFG, n)* **do**
        if *s' $\nsubseteq$ S[n']* **then**
            $S[n'] = S[n'] \cup\ s'$;
            $waitList = waitList \cup \{n'\}$;
        end
    end
end

Algorithm 2: WorkList-Based Abstract Interpretation

trigger because of complex conditions, KLAUS explores symbolic tracing and an instrumentation-based mechanism to bypass these branches in an efficient fashion. In Section 4.5.3, we will elaborate on more design details. It should be noted that considering the complexity of the Linux kernel and the limitation of constraint solvers, we do not use symbolic execution to explore the kernel and thus execute AWRPs. Instead, we leverage more efficient directed fuzzing solutions.

4.4 AWRP Identification

As is mentioned above, the first component of KLAUS is to identify AWRPs introduced by a patch. Here, we introduce how to achieve it by using abstract interpretation. In the following, we first introduce abstract interpretation at a high level. Then, we present our definitions of abstract state, transfer function, and joint operator with regard to AWRP. Finally, we discuss our algorithm to identify AWRPs on the basis of abstract interpretation analysis.

4.4.1 Abstract Interpretation

Abstract interpretation is a static analysis framework that considers all paths and inputs to obtain a sound over-approximation of the state at every program location [141], [142]. For the sake of efficiency, the state is abstract and often represented by a set of constraints in a certain abstract domain. For example, in an interval domain employed in a value-set analysis, each constraint could be in the form of $lb \leq x \leq ub$, where x is a variable and lb, ub are the lower and upper bounds. The joint of two states, $s_1 = lb_1 \leq x \leq ub_1$ and $s_2 = lb_2 \leq x \leq ub_2$ is notated as $s_1 \sqcup s_2 = min(lb_1, lb_2) \leq x \leq max(ub_1, ub_2)$. The $\sqcup$ denotes the join operator which returns an over-approximation of the set union. The goal of using an abstract domain to represent states is to reduce the computational overhead. Therefore, abstract interpretation is especially useful for the analysis of large-scale programs like OS kernel.

Computing on the control flow graph (CFG) of the program, we can reach a fixed-point of states. Without losing generality, in `KLAUS`, we assume the CFG has a unique entry code and a unique exit code. The nodes in CFG are associated with instructions (LLVM IR instructions in our implementation) and edges represent control flows. Algorithm 2 presents a generic worklist-based algorithm to compute the fixed-point. Every node n in the CFG has an abstract state $S[n]$ which is a sound over-approximation of all possible states at n. Initially, $S[n]$ is $\perp$ (empty) for all CFG nodes except the entry node which is $\top$ (tautology). The transfer function in computation takes as an instruction $inst$ and its state s an input, and returns a new state s' which is the result of executing $inst$ in state s. To ensure that analysis converges, when the program has a loop or is non-terminating, a widening operator ∇ is needed in addition to join operator ($\sqcup$). Due to the space limit, we omit details of the widening operator. A more complete introduction can be found in [141], [142].

4.4.2 The Abstract State

In KLAUS, we assume the kernel has a set of variables $V = \{v_1, \cdots, v_n\}$. Each variable $v \in V$ is mapped to $type(v)$ which is a set of annotated types. We use $type(v)$ to construct AWRPs: a write operation and a read operation are paired if the two pertaining variables share the same $type(v)$. More details will be discussed in Section 4.6. Depending on where the variables are stored, $type(v)$ is defined differently.

For local variables the life cycle of which starts at the function prologue and ends at the function epilogue, $type(v)$ = {`'func+x'`} where `'func'` is the function name and `'x'` is the offset of the local variable on the stack frame from the return address. For global or static variables which are valid since kernel bootup and across system calls, we directly use their variable name to denote $type(v)$. That is to say, for the variable like `static struct proto llcp_sock_proto`, its $type(v)$ is `llcp_sock_proto` rather than `struct proto`. For variables stored on the kernel heap (*e.g.,* , SLAB/SLUB region and vmalloc region), we further divided them into three situations. If the variable is an individual object in a structure or union type, we denote it using the type name. If the variable is a structure or union field, we denote its $type(v)$ as `'type+x'` where `'type'` is the type name and `'x'` is the field offset. If the variable is an ordinary buffer, its $type(v)$ is denoted as `'char*'`. In our working example, KLAUS denotes the $type(v)$ of `llcp_sock->local`'s refcount as `nfc_llcp_local+0x18` because the type of `llcp_sock->local` is `nfc_llcp_local` and the offset of the refcount is `0x18`.

Further, each variable $v \in V$ has its value, denoted $value(v)$, which is a set of $\langle cond, content \rangle$ tuples. This tuple indicates that under the condition $cond$, the value of v is equal to $content$. Both $cond$ and $content$ are represented as symbolic strings and KLAUS does not seek to evaluate it using SAT/SMT solvers. In our working example, after the kernel executes line 22, the $value(v)$ set of `llcp_sock->sock` is {⟨ `'llcp_sock->ssap == LLCP_SAP_MAX'`, `'NULL'` ⟩}. For readability, here, we use variable names from the source code. In our implementation, variable names are in SSA (Single

Static Assignment) form.

Applying the above definitions, the abstract state S for our analysis over kernel functions is defined as $S = \{cond, \langle type(v_1), value(v_1)\rangle, \cdots, \langle type(v_n), value(v_n)\rangle\}$ where $cond$ is the path condition accumulated so far from the entry node.

4.4.3 The Transfer Function and Join Operator

To model the impact of executing $inst$ in the state S, we let the transfer function be

$$\text{Transfer}(S, inst).$$

The new state returned by the transfer function is

$$s' = \{cond', \langle type'(v_1), value'(v_1)\rangle, \cdots, \langle type'(v_n), value'(v_n)\rangle\}.$$

If $inst$ writes to a variable v, we update $value(v)$ by removing all tuples and adding a new tuple $\langle cond, content\rangle$ where $cond$ is the path condition in S and $content$ is the writing value. If $inst$ casts variable v from one type to another type, we update $type(v)$ by appending the new type. For any other variables $w \in V$ that are not written or casted, $value(w) = value'(w)$ and $type(w) = type'(w)$. If $inst$ is a conditional jump, $cond$ in S is conjuncted with the jump condition to produce $cond'$. For the instruction $inst$ that neither writes or casts a variable nor performs a conditional jump, $S' = S$.

We define the $Transfer$ for a sequence of instructions $insts = \{inst_0, inst_1, \cdots, inst_n\}$ as $Transfer(S, inst) = Transfer(\cdots(Transfer(S, inst_0), inst_1), \cdots, inst_n)$.

To avoid an exponential increase in state number, states computed along two paths are joined when the control flow merges. For $cond$ in the state, our join operator updates it through disjunction

($cond \cup cond'$). In this way, we can profile the patch condition change introduced by the patch (situation ❹ for AWRPs in Section 4.3.2). Similarly, to capture how variables are affected by the patch and thus construct AWRPs, for each variable $v \in V$, our join operator computes its $type(v)$ and $value(v)$ by combining the sets from two states respectively.

Formally, given two states $S = \{cond, \langle type(v_1), value(v_1)\rangle, \cdots, \langle type(v_n), value(v_n)\rangle\}$ and $S' = \{cond', \langle type'(v_1), value'(v_1)\rangle, \cdots, \langle type'(v_n), value'(v_n)\rangle\}$, we define $S'' = S' \cup S$ as $\{cond \cup cond', \langle type(v_1) \cup type'(v_1), value(v_1) \cup value'(v_1)\rangle, \cdots, \langle type(v_n) \cup type'(v_n), value(v_n) \cup value'(v_n)\rangle\}$.

4.4.4 Identification Algorithm

At a high level, the identification algorithm applies the abstract state, transfer function, and the join operator described above to abstract-interpret the kernel functions before and after patching. By using the differences in analysis results, we identify instructions the state of which are changed by the patch and pair write operation with read operation to construct AWRPs. Algorithm 3 is the sketch of our identification algorithm. In the following, we elaborate on more details.

Intra-procedural analysis. Given a kernel and a patch commit, `KLAUS` can easily find patched functions by parsing the diff file in the commit (line 3 in Algorithm 3). In our working example, the patched function is `llcp_sock_connect` according to line 8 in List 4.1. Our identification starts by applying Algorithm 2 to abstract-interpret these patched functions. More specifically, `KLAUS` analyzes these functions to get S and S' - abstract states before and after patching respectively (line 7-8). For every instruction n in the patched function F, it is an altered instruction if it is newly added in the patch (*i.e.,* , $n \notin F$) or its state changes (*i.e.,* , $S[n] \neq S'[n]$). We deem two states are different as long as the $type(v)$ or $value(v)$ of any one variable v has changed. Note that we do not consider instructions that are deleted by the patch. On the one hand, it is because the triggering

of the new vulnerability needs not execute the deleted instruction. On the other hand, it is because the impact of deleting instructions can be modeled by abstract interpretation and is reflected in the $type(v)$ or $value(v)$ of influenced variables. Finally, we add instructions with altered states to the I set for AWRP construction (line 10-11).

Inter-procedural analysis. The state changes introduced to the patched functions will make impacts on other functions. For a variable v which is passed as an argument to a callee function, if its $value(v)$ changes, all computations in the callee function that depend on this argument will propagate the changes. Similarly, if a function returns a different $value(v)$ to the caller, the computations that use the returned value will change states as well. Therefore, we perform an inter-procedural analysis to further identify these changes. In our algorithm, we use a worklist to maintain functions worth analysis. In line 14 and 17 , we add affected callee and caller functions to the work list until finally, the analysis reaches a fixed-point (line 5).

AWRPs construction. Abstract Interpretation profiles the impact of the patch. Through the above analysis, we obtain a set of instructions I the states of which are altered. According to our discussion in Section 4.3.2, new vulnerabilities result from AWRPs. Therefore, we construct AWRPs from the instruction set I. Recall that the write and read operations in one AWRP pertain to the same variable. Here, we continue to over-approximate the construction. Instead of using memory alias analysis to determine whether two operations belong to the same variable, we examine whether the $type(v)$ of the written variable v and the $type(w)$ of the read variable w have an overlap. If they share a common annotated type, we will pair these two operations. As shown in line 20-21, for any write instruction in the set I, we will search the whole kernel to find a read instruction that satisfies our criteria. In our working example, the type of the variable `llcp_sock->local->ref` in function `llcp_sock_bind` is `nfc_llcp_local+0x18` which is also the type of the variable `sock->local->ref` in function `nfc_llcp_sock_free`. So, we pair the write operation of the former with

the read operation for the latter, and thus construct an AWRP for our output.

4.5 AWRP-Driven Fuzzing

With the identified AWRPs in hand, as mentioned in Section 4.3.3, the second component of KLAUS is designed to perform directed fuzzing. Recall that AWRPs are used to describe the change of control and data flow introduced by a kernel patch. If the patch is problematic, by executing the code relevant to AWRPs, a kernel error could be potentially manifested. In this section, we discuss how KLAUS leverages AWRPs as guidance to search for input that could execute the kernel code pertaining to AWRPs and thus unveil the incorrectness of the corresponding kernel patch.

4.5.1 Two-Dimension Coverage

Before describing the coverage metrics in KLAUS, we need to clarify that AWRP is a necessary but insufficient condition for the new vulnerability. The AWRPs introduced by the patch is intended to fix the old vulnerability. There might not be a new vulnerability. Even if the new vulnerability exists, executing an AWRP does not guarantee the triggering of it unless the variable is assigned with the problematic value (*e.g.,* , a large index to trigger the out-of-bound write) following a certain program path.

Therefore, our fuzzing goal is not only executing identified AWRPs in the order of write and read but also diversifying the writing value and execution path so that we can have more chances to trigger the potential new vulnerability. From this perspective, directed fuzzing that minimizes the distance between the target site and explored site is not an optimal solution for KLAUS. On the one hand, it is because the distance metric overemphasis reaching the target site and fails to enrich variable values along the execution paths. On the other hand, it is because fuzzing that minimizes distance can be easily stuck in a single path and thus is not generalized to consider paths that are

longer but essential to manifesting new vulnerabilities.

Instead of relying on one dominating metric, KLAUS employs a two-dimension coverage strategy to execute code relevant to AWRPs. The first metric is type coverage. Recall that in AWRP identification, the write operation and the read operation are paired if their variables have overlapping in the $type()$ set. Without a doubt, the types in this set are of interest. Favoring inputs that operate kernel objects in these types, KLAUS sticks itself to code that is related to AWRPs so that it does not diverge in the huge kernel codebase. Compared with distance metric, type metric focuses the fuzzing on executing code related to AWRPs and meanwhile allows path exploration. The second metric in KLAUS is code coverage which is commonly used in traditional fuzzing. We do not discard this metric because the new code creates new values that can be assigned to the variables pertaining to AWRPs. With different values, especially extreme values such as a too-large index for buffer access, repeatedly executing write and read operations, KLAUS can ultimately trigger the potential new vulnerability.

4.5.2 Seed Selection

During the course of fuzzing, KLAUS maintains a matrix to filter out interesting seeds. In the matrix, each row represents an identified AWRP. Considering that there will be multiple variable instances that are corresponding to one write operation, in the column, KLAUS stores the pair of the variable address and the writing value.

When an interesting write operation is executed, we compare the address and the value with those already in the matrix. We will add one more pair to the matrix if either the address is new or the value is new. When an interesting read operation is executed, KLAUS examines whether the variable address and the read value exist in the matrix. If not, we will skip this time because to count an AWRP, the writing operation must be executed first.

With this matrix, we assign the highest priority to the input that either executes the interesting write operation with a new writing value or executes the interesting read operation after the corresponding write is executed. Any other inputs that increase type coverage or code coverage are also selected for the following round of fuzzing but not with the highest priority.

4.5.3 Resolving Branch Condition

The design of two-dimension coverage and seed selection keeps the fuzzing on an efficient way of discovering the new vulnerability. However, it does not guarantee that the fuzzing will not be hindered by branch conditions that are hard to satisfy.

Prior works mainly use two approaches to resolve this problem. The first approach is to do symbolic tracing from the system call entry and kick in SAT/SMT solvers to solve the accumulated constraints [125], [143], [144]. The second approach is to instrument the branch and implement a feedback mechanism to favor seeds that bring changes to the variables involved in the branch conditions [120], [145], [146]. However, for large-scale programs like kernel, as we will show in Section 4.7.3, both approaches have their limitations.

For symbolic tracing, though the size and complexity of its constraints are largely reduced compared with symbolic execution, it can go beyond the solvers' capacity from time to time. The major reason is the long calling chain in the kernel. For example, as the kernel is implemented layer by layer when a function in the abstract layer (*e.g.,* , virtual file system) is called, the kernel will execute a series of functions until reaching a low-level layer (*e.g.,* , USB device driver). It will take a very long time for the solver to satisfy the constraints collected in this procedure. In the worst case, the solver might fail to solve the constraints after wasting too much time.

Regarding the instrumentation-feedback mechanism, its main limitation is that it suffers from satisfying complex conditions. It is possible that our targeted write or read operation resides in

a likely-taken branch. So its complex conditions can be easily satisfied with the feedback from instrumentation. However, when it is in a rare branch, the condition can be hard to satisfy. To this end, we experiment with both approaches in KLAUS and thus determine which branch-solving method is more suitable to our problem. We show the evaluation of both methods in Section 4.7.

4.6 Implementation

Our abstract interpretation of the AWRP identification is grounded in the LLVM infrastructure. Our implementation of AWRP-driven fuzzing uses Syzkaller as the foundation. Below, we delve into the specifics of our implementation.

4.6.1 Abstract Interpretation

In Section 4.4, we explain the abstract interpretation at the source code level. The implementation using LLVM brings about certain challenges, which we address below.

First, our AWRP identification process begins with functions that have been altered by the patch. These functions can easily be obtained from the commit. However, when represented at the LLVM IR level, they may have been inlined into their caller functions. To ensure that these caller functions are also considered in our implementation, we utilize `llvm-diff` to identify the patched functions at the LLVM IR level.

Second, KLAUS computes the $type()$ of variables based on the memory region where they are stored. Global variables are easy to identify as their names are unique and present in the LLVM IR. To determine the $type()$ of local variables stored on the stack, KLAUS relies on debugging information to obtain the function names and offsets, which are then combined into the format `'↪ func+x'`. For variables stored in the heap, KLAUS determines the parent structure's type and the variable's offset within the structure using the `Getelementptr` instruction. This instruction is

responsible for getting a field element from a structure using a pointer. Therefore, it has an operand referring to the parent structure and an operand indicating the offset of the field. At the source code level, each variable has its own memory region, making it possible to compute its $type()$ using the above approach. However, in the LLVM IR, some variables are temporary and only exist for the SSA form. In our implementation, we address temporary variables by initially assuming their $type()$ is blank and later computing it as the union of the $type()$ of all operands in the instruction that defines the temporary variable.

Third, `KLAUS` faces a challenge as the LLVM IRs generated from the unpatched and patched kernels are not identical. For example, a variable in the same function may have the label `%var1` ↪ in the IR of the unpatched kernel and `%var2` in the IR of the patched kernel. To accurately identify AWRPs, `KLAUS` needs to know if the $type()$ and $value()$ of a variable have changed due to the patch. To overcome this, `KLAUS` maps variables in both LLVM IRs back to their source code level representation using debugging information. This way, `KLAUS` can compare variables' $type()$ and $value()$ and determine if their corresponding write or read operations should be added to the AWRP set.

Finally, in our implementation, our AWRP identification process first performs an intra-procedural analysis before it spreads the effects of the patch across functions. To handle the `Call` instruction, we treat it as a `Store` operation and consider the called function as a variable. The $value()$ of this pseudo-variable changes if the called function returns a different $value()$. This change is then propagated in an iterative manner, with a list of waiting functions being kept track of, until the $type()$ and $value()$ of all variables reach a fixed point.

4.6.2 AWRP-Driven Fuzzing

Recall that KLAUS experiments two branch-resolving mechanisms to resolve branch conditions – ❶ symbolic tracing, which traces the system call entry symbolically and utilizes an SAT/SMT solver to generate the necessary inputs, and ❷ branch instrumentation, that selects and alters the input seed based on how critical variables are affected.

For symbolic tracing, our implementation follows the method described in HFL [125]. It forks a process specifically for constraint collection and resolution when a branch is touched but has not been entered too many times. For the instrumentation-based mechanism, our implementation instruments critical variables in the kernel during compilation so that the fuzzer can track changes to these variables when the instrumented sites are executed.

In our implementation, two types of variables are treated as critical variables: those directly used in the branch conditions, and those that have a data flow towards the first type of variables. The latter is included in our feedback mechanism because they have the potential to affect the value of the first type of variables.

After capturing changes to the critical variables, our implementation forces the system to revisit how the input was mutated from the previous seed. This allows the system to examine whether the change of the critical variable stems from the mutation of a specific system call argument. If so, this implies an implicit dependence between the argument and the critical variable, and our implementation forks a process to focus on mutating this specific argument to accelerate branch taking.

After KLAUS triggers a kernel error during its directed fuzzing process, it is possible that the error is not caused by the kernel patch under examination. To address this, KLAUS's implementation also includes a triaging component to determine the root cause of the error. If the same error occurs in the unpatched kernel when the patch is removed and the input is replayed, the input is

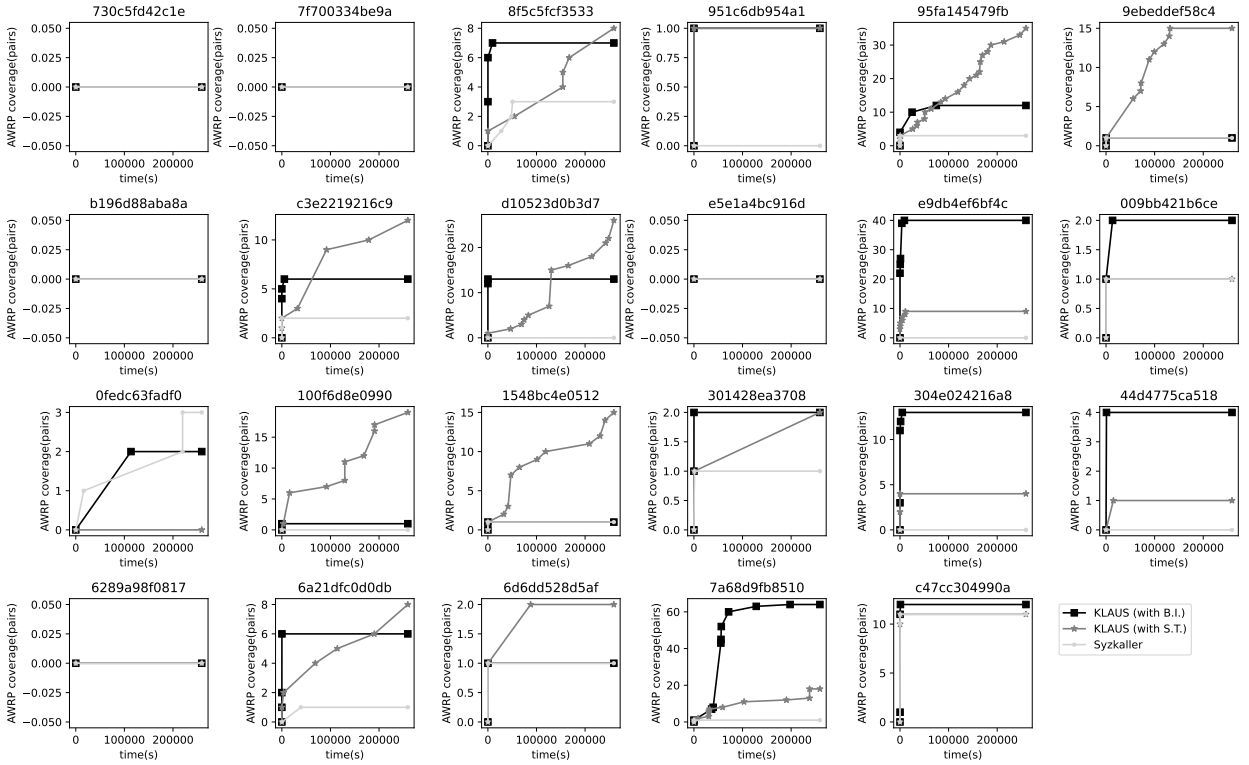

Figure 4.2: The coverage of AWRP across three different kernel fuzzers – Syzkaller, KLAUS implemented with symbolic tracing (denoted by KLAUS with B.I), KLAUS implemented with branch instrumentation (denoted by KLAUS with S.T.). Note that the x-axis represents the time spent on fuzzing, and the y-axis indicates the AWRP coverage.

discarded and treated as not being relevant to the patch. If the error does not occur in the unpatched kernel, we conclude the patch is incorrect. If the error occurs but is different from the error prior to the patch removal, a manual validation procedure is performed to confirm the relevance of the error to the patch. More information on the manual validation process can be found in the Appendix.

4.7 Evaluation

In this section, we first discuss how we construct a ground truth dataset. Then, we describe how to use that dataset to evaluate the effectiveness and efficiency of KLAUS comprehensively and thor-

oughly. Lastly, we demonstrate the utility of KLAUS on more Linux kernel patches (the correctness of which is unknown).

4.7.1 Dataset

Evaluating KLAUS requires a dataset of incorrect Linux kernel patches. Although 3,000+ incorrect patches were gathered and 5% of them were manually analyzed, using them as the ground truth dataset is not feasible. The tool was developed on GCC 9.0 and LLVM 12, limiting the compatibility with some old-version kernels and restricting the available incorrect patches [1]. Also, KLAUS requires the PoC program that triggers the original kernel bug, which was not available for some patches. Thus, only 23 patches could be used as the ground truth dataset. To overcome this shortage, 250 recently released patches were randomly selected for testing. These patches were compatible with the implementation and had available PoC programs for triggering the original bugs. If KLAUS could detect incorrectness in these patches, it would demonstrate the critical utility of the proposed technique.

4.7.2 Experiment Setup

Using the ground truth dataset described above, we compared the effectiveness of KLAUS with that of the most commonly used Linux kernel fuzzing tool – Syzkaller [5]. Besides, we evaluate how the two different branch-resolving mechanisms – instrumentation-based and symbolic tracing-driven solutions – benefit incorrect patch identification. Following this effort, we also study how the two AWRP-based coverage mechanisms each individually contribute to KLAUS's effectiveness in pinpointing incorrect kernel patches.

For all of our experiments, we ran Ubuntu 20.04 LTS on a machine with an AMD EPYC 7702 64-Core CPU and 384GB RAM. For each virtual machine used by our fuzzer, we allocated 2

[1] Note that this is only a limitation of the implementation, our methodology is generic.

virtual CPU cores and 2GB of RAM and ran the fuzzer for 3 days. We collected the results from the fuzzer logs and other files in the fuzzer's work directory. To ensure fairness, we give all fuzzers the same initial seed. Our initial seed was derived from the PoC that triggers the original bug.

4.7.3 Performance in Ground Truth Dataset

Our fuzzing method recorded the time spent on each test case where an error was triggered by an incorrect patch. As seen in Table 4.1, our approach outperforms the traditional kernel fuzzing tool, Syzkaller. By using symbolic tracing and branch instrumentation, respectively, `KLAUS` accurately identified incorrectness in 21 and 23 kernel patches, while Syzkaller only found 13. The reason for this improvement is shown in Figure 4.2. Incorrect patches are often caused by AWRP, and our method showed higher coverage of AWRP, leading to a greater ability to detect patch errors.

As is described in Section 4.5.3, we experiment with two methods to solve branch conditions – symbolic tracing and branch instrumentation. Table 4.1 shows the performance of `KLAUS` when integrating the corresponding approach. As we can observe, `KLAUS` with the integration of branch instrumentation slightly outperforms that with symbolic tracing in terms of the ability to pinpoint incorrect patches (23 vs. 21). In addition, we also observed that, for many test cases, `KLAUS` with branch instrumentation spends less time on exposing the patch incorrectness. The reason is that, when enabling symbolic tracing, `KLAUS` needs more resources to perform constraint solving. While symbolic tracing, to some extent, helps the fuzzer bypass branch predicts more efficiently, it also plays the role of a double-edged sword, slowing down the process by which our fuzzer evaluates inputs. This implies that the branch-instrumentation method is more suitable for kernel patch quality assessment.

In addition to the performance that `KLAUS` exhibits in different branch-solving mechanisms, Table 4.1 also shows `KLAUS`'s performance when it was implemented with other proposed tech-

niques. Recall that, in addition to the two branch-solving schemes, KLAUS is also integrated with an AWRP-based two-dimension coverage mechanism to enhance incorrect patch identification. As is elaborated in Section 4.5.3, the two-dimension coverage scheme contains two parts. One is to utilize the code relevant to AWRP as coverage to guide KLAUS. The other is to employ the type relevant to AWRP as coverage to drive our fuzzer. In our experiment, we break down the impact of each coverage mechanism and show their corresponding performance in Table 4.1.

As we can observe from the table, when disabling branch resolving method and enabling both coverage mechanisms (i.e., utilizing two-dimension coverage alone), KLAUS demonstrates a slight decrease in tracking down incorrect patches (20 vs. 23). It indicates that the branch-resolving mechanism is helpful for improving incorrect patch identification. However, it is not a key driving force for the success of KLAUS.

Looking at the results depicting in the 6^{th} and 7^{th} column of Table 4.1, we can discover that by using the code relevant to AWRP as the coverage guidance alone, KLAUS could exhibit significantly better performance than Syzkaller (18 vs. 13). In contrast, the similar performance gain does not manifest when KLAUS utilizes only the type information pertaining to AWRP as coverage guidance (15 vs. 13). This indicates AWRP-code-based coverage is the key driving force for incorrect patch identification. Besides, the 2^{nd} column implies that AWRP-type-based coverage could be used as a complementary component to improve incorrect patch discovery further.

4.7.4 Performance in the Wild

Recall that the examination of KLAUS' utility involved running it against 250 randomly selected kernel patches with unknown correctness. We discover that KLAUS identified 30 incorrect patches, which we promptly reported to the Linux community. To date, 25 of these patches have been confirmed and fixed. We manually evaluated the bugs introduced by the 30 incorrect patches and found

```
1  struct sock *sock_clone(struct sock *sk) {
2     struct sock *newsk = inet_csk_clone_lock(sk);
3     ...
4     return newsk;
5  }
6
7  int sys_disconnect(struct sock *sk) {
8  | free(sk->uaf);
9  | sk->uaf = NULL;
10 }
11
12 int sys_connect(struct sock *sk) {
13    struct sock *clinet = sock_clone(sk);
14 }
15
16 int sys_close(struct sock *sk) {
17    free(sk->uaf);
18    free(sk);
19 }
20
21 void sk_timer_func(struct sock *sk) {
22    // accessing sk->uaf
23    sk->uaf->a = 1;
24 }
```

Listing 4.3: A simplified kernel code fragment illustrating patch incorrectness. The lines with "--" indicate the code that the patch removes.

that 3 are exploitable across various Linux distributions, including Ubuntu and Android. Interestingly, two of the bugs that the patch intended to fix were not likely to be exploitable, highlighting the danger of incorrect patches which can increase security risks. Due to page limitations, we use only one of the incorrect patches as an example to showcase the change in exploitability.

Case Study. The example in List 4.3 presents a simplified code snippet highlighting the vulnerability caused by an incorrect patch. The function `sock_clone` (line 1) is used to clone a sock object, which includes a pointer `sk->uaf` used for creating a client socket upon new connection establishment (line 13). The function `sys_disconnect` is designed to free `sk->uaf` and set it to null (lines 8 and 9) when the socket is disconnected. In the Linux kernel, freeing a null pointer is acceptable and will not cause any issues.

However, if a timer is set on the socket, the `sk_rest_timer` function will be activated, accessing

`sk->uaf`, which may cause problems if it has already been freed. Imagine a scenario where the timer function is being executed by the kernel while the user closes the socket from the user space, triggering `sys_disconnect` to free `sk->uaf`. In such a case, the timer function will attempt to access the dangling pointer `sk->uaf`, resulting in a use-after-free vulnerability.

The resolution of this vulnerability seemed straightforward initially. Three years ago, kernel developers removed the freeing of `sk->uaf` in `sys_disconnect` (lines 8 and 9) to eliminate the race condition, as reflected in the code snippet. However, upon testing the patch using KLAUS, it was discovered to be incorrect, and the bug introduced by this patch was even more susceptible to exploitation.

By delving into KLAUS running against the simple patch above, we observe that KLAUS first detects the altered variable `sk->uaf` from deleted lines 8 and 9. Using the type information associated with `sk->uaf`, KLAUS then conducts inter-procedure analysis and identifies related variables in functions `sock_clone`, `sys_close`, and `sk_timer_func`. These identified variables together form the AWRPs, guiding KLAUS's fuzzing component to proceed as follows.

First, the fuzzer executes `sys_disconnect`, causing the altered write on `sk->uaf`. The annotated pair then directs the fuzzer to run the `sk_clone` function, where the copy of `sk->uaf` is made. After the socket is closed, executing `sys_close` and freeing `sk->uaf`'s memory, the leftover socket's `sk->uaf` becomes a dangling pointer. Compared with the dangling pointer obtained through the original kernel bug, the dangling pointer caused by the incorrect patch is more stably exploitable because it no longer requires a critical race to trigger the use-after-free, as discussed.

4.8 Related Work

In this work, we analyze Linux kernel patches and employ directed fuzzing techniques to pinpoint the ones developed incorrectly. To this end, the works most relevant to ours are patch correctness

analysis and directed fuzzing.

Patch correctness analysis. Prior works have explored various methods to evaluate the quality of bug patches. For example, Gu *et al.* proposed Fixation, a system that uses distance-bounded weakest precondition to identify partially fixed exceptions in Java programs [170]. Kim *et al.* introduced the concept of bug neighborhood to check for persistence of NULL pointer references after patch application [171]. Le and Pattison proposed a new program representation, the multi-version interprocedural control flow graph, which describes control flow variations between different software versions. By performing path-sensitive symbolic analysis on the graph, they showed that their representation could efficiently verify patches [172].

The prior works mentioned are not applicable to our problem. They were designed to address a specific error (e.g. pinpointing exceptions in Java programs [170] or NULL pointer references [171]), while our goal is to uncover incorrect patches for various memory corruption errors. The prior research focuses on examining the completeness of the given patch, but our aim is to not only identify partially fixed bugs but also to expose new bugs or vulnerabilities introduced by the patch. Lastly, the prior research focuses on examining problematic bug fixes in userspace programs, whereas our goal is to unveil incorrect kernel patches. The complexity of the kernel limits the utility of many program analysis techniques (such as symbolic execution adopted in [172]).

Directed Fuzzing. In the past, various directed fuzzing methods have been introduced. AFLGo [127] prioritizes the input seed with the shortest path to the target program site at the basic block level. Hawkeye [173] calculates the similarity between the input seed and the potential execution trace, and uses this similarity to guide the selection of input seeds. To perform directed fuzzing on the Linux kernel, SemFuzz [126] combines NLP techniques with program analysis. KATCH [174] selects the input with the shortest distance to the target, and then uses symbolic execution. Savior [175] only employs symbolic execution if it visits branches that can reach targets with poten-

tially buggy code. BEACON [176] calculates preconditions for reaching the target and prunes paths that cannot reach the target, avoiding wasted computational resources. FuzzGuard [177] trains a predictor classifier and uses it to prioritize inputs more likely to reach bugs. Regression greybox fuzzing [178] proposes a new power scheduling mechanism, favoring seeds that cover frequently changing code.

In contrast to prior directed fuzzing techniques, our approach utilizes a distinct feedback mechanism to guide the fuzzer towards the target code. Our directed fuzzing method prioritizes not just the reachability of the target code, but also diversifying the paths towards it, resulting in a wider exploration of execution contexts. Additionally, our directed fuzzing takes into account the sequential reachability of the code.

4.9 Conclusion

The Linux kernel, with its complex structure and intricate system calls, presents a significant challenge for developers looking to create patches for bugs. Incorrect patches can be problematic, transforming a non-exploitable bug into a highly exploitable vulnerability. This is due in part to the limitations of existing tools and techniques for identifying incorrect patches. Our research discovers that a kernel fuzzer, which uses alteration to the variables' read-write operations as a guide, has the potential to be a more effective tool for pinpointing incorrect patches. Besides, our research also uncovers that by incorporating existing branching-resolving mechanisms, the ability of the kernel fuzzer to identify incorrect patches can be further enhanced. With these discoveries in hand, we believe that an automated method for tracking down incorrect kernel patches could greatly improve the overall quality of patches for the Linux kernel. As a next step in our research, we plan to explore the identification of incorrect patches for userland programs. This will help to ensure the reliability and security of the Linux kernel and its userland programs, a crucial compo-

nent of modern computing.

Input: Kernel K and Patch P ;
Output: Pair[] ;
$funcWL$ = GetPatchedFunc(K, P) ;
I = {}, $Pair$ = {} ;
while ∃ *¡F, F'¿* ∈ *funcWL* **do**
    $funcWL$ = $funcWL$ - {¡F, F'¿} ;
    S = AbstractInterpret(F) ;
    S' = AbstractInterpret(F') ;
    foreach $n \in F'$ **do**
        if *n* ∉ *F or S[n]* ≠ *S'[n]* **then**
            I = $I \cup$ {n'} ;
        end
        if *IsCall(n)* **then**
            if ∃ *arg, value(arg)* ≠ *value'(arg)* **then**
                $funcWL$ = $funcWL \cup$ Callee(n) ;
            end
        end
        if *IsRet(n)* **then**
            if *value(ret)* ≠ *value'(ret)* **then**
                $funcWL$ = $funcWL \cup$ Caller(n) ;
            end
        end
    end
end
foreach $n \in I$ **do**
    if *IsWrite(n)* **then**
        m = FindRead($type(n)$, K) ;
        $Pair$ = $Pair \cup$ ¡n, m¿ ;
    end
end

Algorithm 3: Sketch of AWRPs Identifcation

Case ID	KLAUS $_{B.I.}$	Syzkaller	KLAUS $_{N.R.}$	KLAUS $_{S.T.}$	KLAUS $_{B.I.}$(code only)	KLAUS $_{B.I.}$(type only)
730c5fd42c1e[147]	6m	18m	37m	9m	240m	¡ 1m
7f700334be9a[148]	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
8f5c5fcf3533[149]	2m	N/A	1m	¡ 1m	2m	N/A
951c6db954a1[150]	180m	480m	25m	955m	360m	N/A
95fa145479fb[151]	1080m	N/A	1902m	2989m	N/A	N/A
9ebeddef58c4[152]	44m	1680m	60m	119m	60m	N/A
b196d88aba8a[153]	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
c3e2219216c9[154]	¡ 1m	4260m	4m	1535m	N/A	9m
d10523d0b3d7[155]	4m	3960m	8m	7m	4m	9m
e5e1a4bc916d[156]	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
e9db4ef6bf4c[157]	720m	N/A	N/A	51m	N/A	N/A
009bb421b6ce[158]	¡ 1m	N/A	¡ 1m	¡ 1m	¡ 1m	N/A
0fedc63fadf0[159]	2665m	N/A	N/A	N/A	3518m	3800m
100f6d8e0990[160]	¡ 1m	1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
1548bc4e0512[161]	451m	N/A	199m	N/A	1m	3488m
301428ea3708[162]	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
304e024216a8[163]	21m	N/A	3m	1m	¡ 1m	62m
44d4775ca518[164]	1835m	N/A	N/A	4212m	N/A	N/A
6289a98f0817[165]	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m	¡ 1m
6a21dfc0d0db[166]	3m	1200m	1m	62m	5m	N/A
6d6dd528d5af[167]	592m	N/A	54m	86m	2904m	1141m
7a68d9fb8510[168]	4116m	N/A	657m	3474m	N/A	658m
c47cc304990a[169]	24m	20m	17m	3m	6m	7m
total #	23	13	20	21	18	15

Table 4.1: The performance of KLAUS. Note that the case ID represents the commit ID of the incorrect patch. The number in the table indicates the time spent for a corresponding fuzzing method to pinpoint an incorrect patch in minutes. "N/A" denotes that the fuzzing method fails to find the incorrectness within 3 days. The last row shows the total number of incorrect patches discovered within this time period. KLAUS $_{B.I.}$ and KLAUS $_{S.T.}$ denote the versions of our fuzzer using Branch Instrumentation and Symbolic Tracking, respectively. KLAUS $_{N.R.}$ runs without branch resolving. KLAUS $_{B.I.}$(type only) and KLAUS $_{B.I.}$(code only) use only type-related or code-relevant AWRP guidance.

CHAPTER 5
CONCLUSION

Enhance Fuzzing Ecosystem

This dissertation strengthens the modern fuzzing ecosystem by addressing three core stages of the vulnerability discovery pipeline: seed generation, bug report management, and patch validation. They form a continuous workflow whose overall effectiveness is determined by the weakest link. The works presented here demonstrate how compiler instrumentation, program analysis, and automated reasoning can systematically enhance each stage and make large-scale fuzzing more reliable and scalable.

First, we automated the pre-fuzzing stage by designing an LLM-guided seed generation framework that extracts input constraints directly from program logic. Through compiler instrumentation and static analysis, our system isolates validation-relevant code, and an LLM agent, combined with a code search engine, infers precise input grammars. Integrated with Grammar Mutator and Jazzer, these grammars generate high-quality seeds that significantly reduce time-to-bug discovery and contribute to our success in the DARPA AI Cyber Challenge.

Second, we improved the mid-fuzzing stage by conducting the first large-scale empirical study of kernel bug report duplication, revealing that nearly half of Syzkaller/Syzbot crash reports stem from the same underlying bugs. By identifying key root causes of duplication and developing targeted detection strategies, we substantially enhanced the accuracy of deduplication, reducing developer burden and improving the efficiency of continuous fuzzing workflows.

Finally, we addressed the post-fuzzing stage with a structure-guided patch validation frame-

work that analyzes patch-induced changes to memory access patterns and directs fuzzing toward the most relevant code regions. Our system uncovers incorrect and incomplete Linux kernel patches that evade existing methods, reinforcing the reliability of the final steps in the vulnerability remediation process.

Together, these contributions present a unified and cohesive advancement of the fuzzing ecosystem, automating how fuzzing begins, managing what fuzzing produces, and validating what fuzzing ultimately enables, thereby improving the end-to-end robustness of modern vulnerability discovery pipelines.

Futrue Plan

There remain several promising directions that can further strengthen the automation and reliability of the fuzzing ecosystem. One direction is to explore adaptive, self-improving seed generation, where input grammars continually refine themselves based on dynamic feedback gathered during fuzzing. Another avenue lies in developing more generalizable mechanisms for understanding program semantics, beyond syntactic validation, to enable fuzzers to reason about stateful protocols, multi-stage inputs, and environment-dependent behaviors. Additionally, as large-scale fuzzing deployments increasingly rely on continuous integration pipelines, integrating automated patch validation and failure triage into upstream development workflows presents an important challenge. Advancing these directions will help push modern fuzzing systems toward a fully autonomous, end-to-end vulnerability discovery and remediation pipeline.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," in *USENIX Summer*, 1990, pp. 1–13.

[2] M. Zalewski, *American fuzzy lop (afl)*, https://lcamtuf.coredump.cx/afl/, 2014.

[3] A. Fioraldi, D. Maier, S. Schumilo, and C. Aschermann, "Afl++: Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[4] K. Serebryany, "Continuous fuzzing with libfuzzer," in *USENIX Security Symposium*, 2016.

[5] Google, *Syzkaller: Kernel fuzzer*, https://github.com/google/syzkaller, 2017.

[6] Google, *Oss-fuzz: Continuous fuzzing for open source software*, https://github.com/google/oss-fuzz, 2016.

[7] Google, *Clusterfuzz*, https://google.github.io/clusterfuzz/, 2019.

[8] Google, *Syzbot: Automated kernel bug tracker*, https://syzkaller.appspot.com/, 2017.

[9] M. Pérennès, L. Peyton, et al., "The art, science, and engineering of fuzzing: A survey," in *IEEE Transactions on Software Engineering*, 2018.

[10] R. Paleari et al., "Uncovering duplicate bugs in fuzzing: A large-scale empirical study," in *NDSS Symposium*, 2021.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating systems errors," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 73–88.

[12] DARPA, *Darpa ai cyber challenge (aixcc)*, https://aicyberchallenge.org/, 2023.

[13] DARPA, *Aixcc finalist teams announced*, https://aicyberchallenge.org/finalists, 2024.

[14] D. Mu et al., "An in-depth analysis of duplicated linux kernel bug reports," in *Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[15] Y. Wu, Z. Lin, Y. Chen, D. K. Le, D. Mu, and X. Xing, "Mitigating security risks in linux with KLAUS: A method for evaluating patch correctness," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 4247–4264, ISBN: 978-1-939133-37-3.

[16] R. Ford and J. Cappaert, "The growing complexity of input formats in modern software," in *IEEE Security & Privacy Magazine*, 2019.

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing: Brute force vulnerability discovery," in *USENIX Security*, 2012.

[18] I. S. Research, *Laf-intel: Structure-aware fuzzing enhancements*, https://software.intel.com, 2017.

[19] S. Han et al., "Neuzz: Efficient fuzzing with neural program smoothing," in *IEEE S&P*, 2022.

[20] S. Gan et al., "Griffin: Grammar-aware fuzzing for network protocols," in *USENIX Security*, 2020.

[21] V. Pham et al., "Smartseed: Quality seed synthesis for fuzzing," in *USENIX Security*, 2019.

[22] M. Chen et al., "Evaluating large language models trained on code," in *arXiv preprint arXiv:2107.03374*, 2021.

[23] B. Fu et al., "Llmfuzz: Fuzzing with large language models," in *USENIX Security*, 2024.

[24] C. I. GmbH, *Jazzer: Coverage-guided fuzzing for the jvm*, https://github.com/CodeIntelligenceTesting/jazzer, 2023.

[25] B. Livshits and M. Lam, "Finding security bugs in java applications with static analysis," in *USENIX Security*, 2005.

[26] G. F. Team, *Grammar mutator specification*, https://github.com/google/oss-fuzz/tree/master/src/grammar-mutator, 2020.

[27] DARPA, *Darpa aixcc finalist announcement*, https://aicyberchallenge.org/finalists, 2024.

[28] G. S. Team, *Grammar mutator*, https://github.com/google/oss-fuzz/tree/master/src/grammar-mutator, 2020.

[29] O. C. (contributors), *Opengrok: Source code search and cross reference engine*, https://github.com/oracle/opengrok, 2024.

[30] G. P. Zero and G. O.-F. Team, *Jazzer: Coverage-guided fuzzing for the jvm*, https://github.com/CodeIntelligenceTesting/jazzer, 2021.

[31] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[32] F. Hsu, Y. Xie, and et al., "Polyglot: Automatic extraction of protocol message formats from network traces," in *USENIX Security Symposium*, 2020.

[33] M. Hänggi and M. Pradel, "Autogram: Automatic extraction of input grammars from software," in *Proceedings of ESEC/FSE*, 2018.

[34] D. Zhu, J. Choi, and et al., "Digger: Dynamic grammar recovery for real-world parsers," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[35] C. Holler, K. Herzig, and A. Zeller, "Langfuzz: Fuzzing for code coverage," in *Proceedings of PLDI*, 2012.

[36] C. Aschermann, S. Schumilo, and et al., "Nautilus: Fishing for deep bugs with grammars," in *NDSS Symposium*, 2019.

[37] H. Wang, W. Chen, and et al., "Superion: Grammar-aware greybox fuzzing," in *ICSE*, 2019.

[38] M. Chen, J. Tworek, et al., *Evaluating large language models trained on code*, https://arxiv.org/abs/2107.03374, 2021.

[39] Z. Feng and et al., "Codebert: A pre-trained model for programming and natural languages," in *EMNLP*, 2020.

[40] P. Ding and et al., "Fuzzgpt: Leveraging large language models for enhanced fuzzing," in *USENIX Security Workshop*, 2023.

[41] Y. Zhao and et al., *Llm-assisted mutation strategies for fuzzing*, https://arxiv.org/abs/2308.04270, 2023.

[42] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11, 2011.

[43] *WannaCry Ransomware Attack*, https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2017.

[44] *Dirty COW (CVE-2016-5195)*, https://dirtycow.ninja/, 2016.

[45] *Bleedingtooth*, https://securityaffairs.co/wordpress/109500/hacking/bluetooth-bleedingtooth-vulnerabilities.html, 2020.

[46] *Trinity: Linux system call fuzzer*. https://github.com/kernelslacker/trinity, 2006.

[47] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26rd USENIX Conference on Security Symposium*, ser. USENIX SEC '17, 2017.

[48] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '20, 2020.

[49] CVE Details, *Linux Kernel*, https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2021.

[50] T. Blazytko et al., "AURORA: Statistical crash analysis for automated root cause explanation," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[51] *Fault injection capabilities infrastructure*, https://www.kernel.org/doc/html/latest/fault-injection/fault-injection.html, 2006.

[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC '12, 2012.

[53] *Kernel address sanitizer*, https://www.kernel.org/doc/html/latest/dev-tools/kasan.html, 2014.

[54] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15, 2015.

[55] *Kernel memory sanitizer*, https://github.com/google/kmsan, 2015.

[56] *The kernel concurrency sanitizer (kcsan)*, https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html, 2019.

[57] *The undefined behavior sanitizer - ubsan*, https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html, 2021.

[58] *Kernel memory leak detector*, https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html, 2011.

[59] *The object-lifetime debugging infrastructure*, https://www.kernel.org/doc/html/latest/core-api/debug-objects.html, 2016.

[60] D. Mu et al., "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, ser. USENIX SEC '18, 2018.

[61] D. Borkmann, *Bpf, array: Fix overflow in max_entries and undefined behavior in index_mask*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bbeb6e4323dad9b5e0ee9f60c223dd532e2403b1, 2018.

[62] *New bug left in the closed and duplicated bug reports*, https://groups.google.com/g/syzkaller/c/_roBlPUWp04/m/Na_lAPP7AwAJ, 2021.

[63] *memory leak in hub_event*, https://syzkaller.appspot.com/bug?id=66fe8eb71f455a245547576eb8d36fec957d2424, 2021.

[64] *memory leak in hub_event (2)*, https://syzkaller.appspot.com/bug?id=91adc3631f0fd2b51356949a8cc20b994856d6ee, 2021.

[65] *Common Vulnerabilities and Exposures (CVE)*, https://cve.mitre.org/, 1999.

[66] L. Kernel, *Submitting patches: The essential guide to getting your code into the kernel*, https://www.kernel.org/doc/html/v4.10/process/submitting-patches.html, 2021.

[67] ClusterFuzz, *Crash type in clusterfuzz*, https://google.github.io/clusterfuzz/reference/glossary/crash-type, 2019.

[68] J. Foote, *The exploitable gdb plugin*, https://github.com/jfoote/exploitable, 2015.

[69] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, ser. SP '20, 2020.

[70] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[71] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables.," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[72] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity.," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '16, 2016.

[73] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. USENIX SEC '14, 2014.

[74] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. USENIX SEC '18, 2018.

[75] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. USENIX SEC '19, 2019.

[76] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[77] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014.

[78] Z. Jiang et al., "PDiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[79] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019.

[80] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[81] Z. Xu et al., "Automatic hot patch generation for android kernels," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[82] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. USENIX ATC '05, 2005.

[83] S. dashboard, *KASAN: Use-after-free read in map_lookup_elem*, https://syzkaller.appspot.com/bug?id=fcd138b2ad0188e5eed65d3351ab983f4bc1c3b6, 2018.

[84] S. dashboard, *KASAN: Use-after-free write*, https://syzkaller.appspot.com/bug?id=19cf067af88c8f151825cefa7b199e7a8b7dc861, 2018.

[85] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *10th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '12, 2012.

[86] A. Stern, *HID: Hidraw: Fix invalid read in hidraw_ioctl*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=416dacb819f59180e4d86a5550052033ebb6d72c, 2019.

[87] E. Dumazet, *Macvlan: Do not assume mac_header is set in macvlan_broadcast()*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=96cc4b69581db68efc9749ef32e9cf8e0160c509, 2018.

[88] Z. Mithra, *Apparmor: Fix uninitialized value in aa_split_fqname*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=250f2da49cb8e582215a65c03f50e8ddf5cd119c, 2018.

[89] S. dashboard, *KASAN: Slab-out-of-bounds read in hidraw_ioctl*, https://syzkaller.appspot.com/bug?id=0141bd6b37153edec9c4ffa0f0e990c7228897f9, 2019.

[90] S. dashboard, *KMSAN: Use-after-free in hidraw_ioctl*, https://syzkaller.appspot.com/bug?id=572cfde68d72a02f41e60c6a6c060157c6e6a750, 2019.

[91] googlegroup, *KASAN: Slab-out-of-bounds read in hidraw_ioctl*, https://groups.google.com/g/syzkaller-bugs/c/O90aBzvp_uE/m/-Q0j14aUBwAJ, 2019.

[92] *Mm: Kasan: Initial memory quarantine implementation*, https://lore.kernel.org/patchwork/patch/658546/, 2016.

[93] W. Zhang et al., "Conseq: Detecting concurrency bugs through sequential errors," *SIGPLAN Not.*, vol. 46, no. 3, pp. 251–264, Mar. 2011.

[94] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019.

[95] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th USENIX Conference on Security Symposium*, ser. USENIX SEC '09, 2009.

[96] *Systemtap*, https://sourceware.org/systemtap/, 2005.

[97] S. dashboard, *Gpf in ip_sublist_rcv*, https://syzkaller.appspot.com/bug?id=f07ddeedf116ac76456528915123a4d8d4d709bd, 2019.

[98] S. dashboard, *General protection fault in ip6_sublist_rcv*, https://syzkaller.appspot.com/bug?id=cdfd70388a396eb80fe860a5251c2e1232b1e407, 2019.

[99] S. dashboard, *KASAN: Invalid-free in disconnect_rio (2)*, https://syzkaller.appspot.com/bug?id=35acc31cfe715b32b649129f286c960dba2d49c5, 2019.

[100] S. dashboard, *General protection fault in flexcop_usb_probe*, https://syzkaller.appspot.com/bug?id=c0203bd72037d07493f4b7562411e4f5f4553a8f, 2019.

[101] S. dashboard, *UBSAN: Shift-out-of-bounds in mceusb_dev_recv*, https://syzkaller.appspot.com/bug?id=50d4123e6132c9563297ecad0479eaad7480c172, 2020.

[102] S. dashboard, *UBSAN: Shift-out-of-bounds*, https://syzkaller.appspot.com/bug?id=df1efbbf75149f5853ecff1938ffd3134f269119, 2020.

[103] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing.," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[104] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013.

[105] CERT, *BFF - basic fuzzing framework*, https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework, 2016.

[106] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007.

[107] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE'08, 2008.

[108] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012.

[109] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[110] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[111] C. Lyu et al., "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. USENIX SEC '19, 2019.

[112] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, ser. SP '18, 2018.

[113] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[114] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. FSE '17, 2017.

[115] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[116] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[117] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'17, 2017.

[118] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with two-dimensional consistency checking," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[119] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *Proceedings of The Network and Distributed System Security Symposium*, 2021.

[120] Z. Lin et al., "Grebe: Facilitating security assessment for linux kernel bugs," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[121] Y. Zhai et al., "Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[122] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[123] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning.," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[124] B. Zhao et al., "StateFuzz: System Call-Based State-Aware linux driver fuzzing," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[125] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel.," in *Proceedings of The Network and Distributed System Security Symposium*, 2020.

[126] W. You et al., "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[127] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[128] *Pwn2Own*, https://en.wikipedia.org/wiki/Pwn2Own.

[129] *kCTF*, https://google.github.io/kctf/.

[130] *Linux kernel selftests*, https://www.kernel.org/doc/Documentation/kselftest.txt.

[131] *0-day CI*, https://www.intel.com/content/www/us/en/developer/articles/technical/0-day-ci-linux-kernel-performance-report-v5-3.html.

[132] *Kernel CI*, https://kernelci.automotivelinux.org/.

[133] *Syztest*, https://github.com/google/syzkaller/tree/master/pkg/runtest.

[134] Q. Wu, Y. Xiao, X. Liao, and K. Lu, "OS-Aware vulnerability prioritization via differential severity analysis," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[135] *CVE-2021-23134*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23134.

[136] *Net/nfc: Fix use-after-free llcp_sock_bind/connect*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c61760e6940d.

[137] *Nfc: Fix refcount leak in llcp_sock_connect*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8a4cd82d62b5.

[138] *Syzkaller dashborad*, https://syzkaller.appspot.com/.

[139] H. Wang et al., "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[140] H. Kim et al., "Patchverif: Discovering faulty patches in robotic vehicles," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[141] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

[142] A. Miné, "The octagon abstract domain," in *Proceedings of the 8th Working Conference on Reverse Engineering*, 2001.

[143] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[144] K. Kim et al., "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[145] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[146] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[147] *Rxrpc: Fix local endpoint refcounting*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=730c5fd42c1e.

[148] *Ip6_gre: Proper dev_{hold—put} in ndo_[un]init methods*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7f700334be9a.

[149] *Tipc: Call start and done ops directly in __tipc_nl_compat_dumpit*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8f5c5fcf3533.

[150] *Sctp: Fix memleak on err handling of stream initialization*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=951c6db954a1.

[151] *Bpf: Sockmap/tls, close can race with map free*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95fa145479fb.

[152] *Rxrpc: Rxrpc_peer needs to hold a ref on the rxrpc_local record*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ebeddef58c4.

[153] *Tun: Fix use after free for ptr_ring*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b196d88aba8a.

[154] *Block: Free sched's request pool in blk_cleanup_queue*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c3e2219216c9.

[155] *Fix d1052*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d10523d0b3d7.

[156] *Fix e5e1a*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e5e1a4bc916d.

[157] *Bpf: Sockhash fix omitted bucket lock in sock_close*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e9db4ef6bf4c.

[158] *Workqueue, lockdep: Fix an alloc_workqueue error path*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=009bb421b6ce.

[159] *Fix 0fedc*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0fedc63fadf0.

[160] *Net: Correct zerocopy refcnt with udp msg_more*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=100f6d8e0990.

[161] *Xfrm: Policy: Delete inexact policies from inexact list on hash rebuild*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1548bc4e0512.

[162] *Net/smc: Fix refcounting for non-blocking connect*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=301428ea3708.

[163] *Commit 304e024216a8*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=304e024216a8.

[164] *Net/sched: Sch_taprio: Reset child qdiscs before freeing them*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=44d4775ca518.

[165] *Sit: Proper dev_hold—put in ndo_[un]init methods*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6289a98f0817.

[166] *Rdma/ucma: Limit possible option size*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6a21dfc0d0db.

[167] *Net/smc: Fix refcount non-blocking connect -part 2*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6d6dd528d5af.

[168] *Usb: Usbdevfs: Sanitize flags more*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7a68d9fb8510.

[169] *Net: Kcm: Fix memory leak*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c47cc304990a.

[170] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[171] M. Kim, S. Sinha, C. Görg, H. Shah, M. J. Harrold, and M. G. Nanda, "Automated bug neighborhood analysis for identifying incomplete bug fixes," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010.

[172] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[173] H. Chen et al., "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[174] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[175] Y. Chen et al., "Savior: Towards bug-driven hybrid testing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[176] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[177] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Filtering out unreachable inputs in directed greybox fuzzing through deep learning," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[178] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[179] S. Santoyo, *Outlier detection*, https://towardsdatascience.com/a-brief-overview-of-outlier-detection-techniques-1e0b2c19e561, 2017.

APPENDIX A

APPENDIX

Appendix-A: Corner Cases of Bug Groups

Our "ground-truth" bug groups are determined based on the patches of those fixed bugs. This methodology is reliable, except for a handful of corner cases. We discover such corner cases during our manual analysis, which turns out to be caused by human errors of syzbot maintainers. More specifically, thorough our analysis (described in Section 3.3.4), we find that 5 bug groups (4.59%) contain the wrong patches for 10 of their bugs. Note that this is not a contributing factor to bug duplication (but may have some relationship with duplication). More specifically, after a bugfix is developed, certain syzbot maintainers may manually assign the bugfix to one or more bug titles since they expect there are duplications. We find that sometimes the patch assignment is wrong. The mistake is mostly caused by human errors during their manual bug analysis (e.g., recognizing the wrong bug in the same parts of the code). To mitigate this issue, we recommend simply re-running the PoC to verify if the patch is indeed fixing for the assigned bug. We have reported this problem to the corresponding syzbot maintainers.

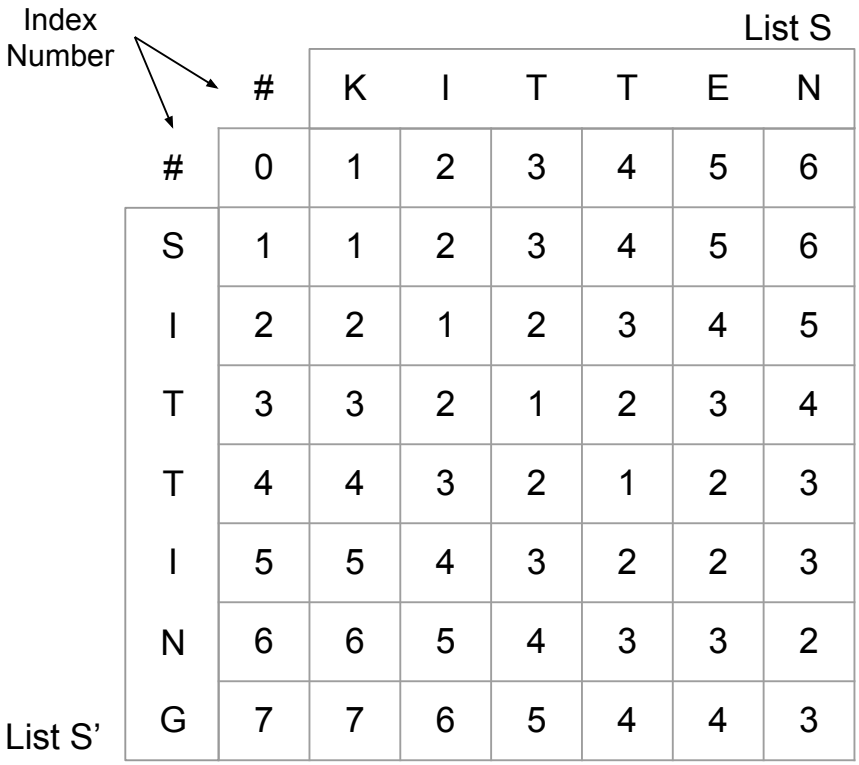

Figure A.1: A sample matrix obtained from Levenshtein distance computation.

Appendix-B: Similarity Measure

Levenshtein distance. The Levenshtein distance equation takes as input two lists, outputting a matrix by using the equation below

$$Lev_{S,S'}(i,j) = \begin{cases} max(i,j), & \text{if } min(i,j) = 0. \\ min \begin{cases} Lev_{S,S'}(i-1,j) + 1, \\ Lev_{S,S'}(i,j-1) + 1, & \text{Otherwise.} \\ Lev_{S,S'}(i-1,j-1) + cost(S_i, S'_j), \end{cases} \end{cases} \tag{A.1}$$

As is depicted in Figure A.1, the Levenshtein distance computation can identify the element aligned across the lists. In our application, they indicate the syscalls aligned across the pair of PoC.

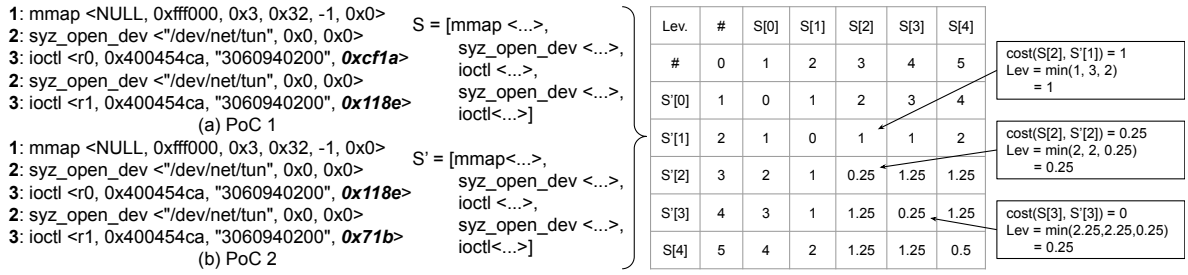

Figure A.2: The demonstration of our customized Levenshtein distance computation.

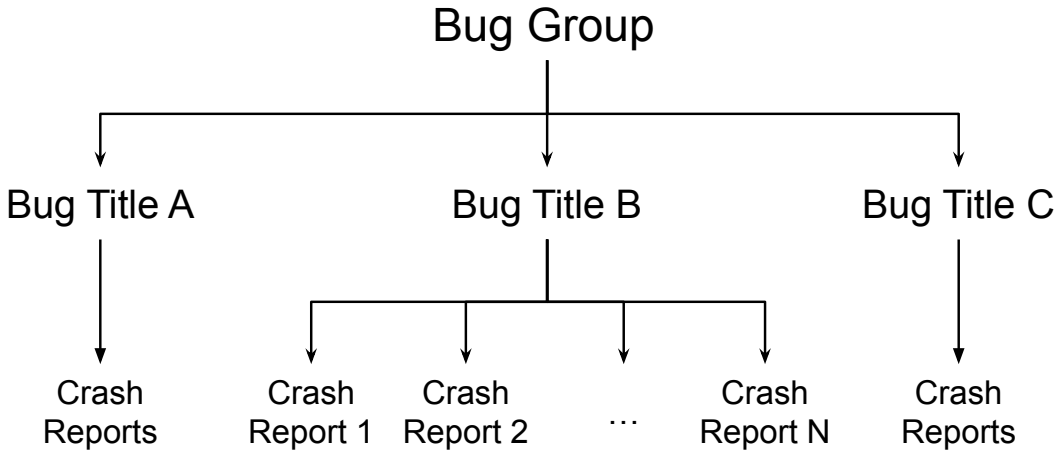

Figure A.3: Relationships between bug groups, bug titles, and crash reports.

In addition, the value in the lower-right corner of the matrix also indicates the minimal edit needed for flipping one list into the other. It is in a range from 0 to ∞. In our work, we normalize this value into the range between 0 and 1 by using the equation below.

$$Similarity(S, S') = 1 - \frac{Lev(S, S')}{max(len(S), len(S'))} \tag{A.2}$$

We use this normalized value as the similarity between the two lists extracted from the PoCs.

Customized Levenshtein distance. Different from the original Levenshtein distance equation, our customized version replaces the cost function of the original version from the form $cost(a_i, b_j) =$

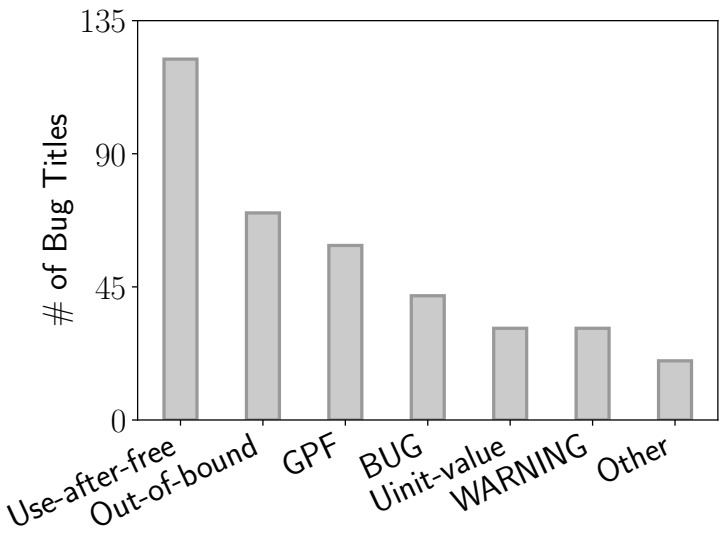

Figure A.4: Number of bug titles under each distinct crash type in our *sampled* dataset.

$1_{a_i!=b_j}; 0_{a_i==b_j}$ to the form below

$$cost(S_i, S_j) = \begin{cases} 1 & ID(S_i) \neq ID(S_j), \\ N/M & ID(S_i) = ID(S_j) \end{cases} \tag{A.3}$$

Here, the new cost function is designed to augment the Levenshtein distance equation with the ability to capture the argument deviation of system calls in the same family. As we can observe from Figure A.2, if the syscall aligned in both PoC have the exact same arguments, we assign 0 to the cost function. Otherwise, we deem $\frac{N}{M}$ as the value of the cost function. Here, N represents the total number of arguments sharing no value whereas M indicates the total number of arguments in the syscall's argument list.

Threshold selection. To determine what value of the similarity measure should be taken as a signal for duplicated report identification, we use the customized Levenshtein distance to compute the pairwise similarity for non-duplicated kernel reports gathered from Syzbot. We note the average pairwise similarity measure for these non-duplicated reports is 0.16, and their standard deviation

is 0.14. Based on the univariant outlier detection technique [179], we use the mean value plus the 3.5x standard deviation as our cutoff threshold value. We deem a pair of reports have a highly similar PoC program if their similarity measure is above this threshold.

Appendix-C: Detection Performance of Bug Groups

The evaluation of the proposed techniques (Section 3.6.3) has been focused on the detection of duplicated *bug pairs*. Here, we briefly discuss the detection performance at the *bug group* level. More specifically, we grouped the detected duplicated pairs into bug groups and compared them with the ground truth. For our proposed method, we find that the detection has 0 (zero) error for 70 bug groups (out of 104 ground-truth groups). This means our techniques have perfectly discovered these bug groups. In comparison, the baseline method only discovered 44 bug groups without any errors. At the same time, our method only produces 5 bug groups that contain false positives (10 FP pairs in total). In comparison, the baseline method has produced 70 bug groups with false positives in the group. The rest of the bug groups do not have false positives, but may have missed duplicated bug titles. Overall, the results confirm that our method significantly outperforms the existing baseline method. Further discussion on how false positives should be handled is in Section 3.6.3.

Appendix-D: Implementation of the baseline method

As is mentioned in Section 3.6, when grouping duplicated bug reports gathered from Syzbot, we re-implemented the method used by ClusterFuzz. In this work, we used this method as our comparison baseline to demonstrate the effectiveness of our proposed deduplication method. Here, we detail our re-implementation effort. Given a pair of Linux kernel bug reports, we first extracted the crashing stack trace from each report. In this step, we customized ClusterFuzz's deduplication

method by reconsidering the format of Linux kernel stack trace and kernel sanitizers. Meanwhile, we excluded certain functions at the kernel stack trace (*e.g.,* `dump_stack`) to prevent its adverse effect upon the similarity comparison of the stack traces. Second, we then compared the stack traces of the two bug reports one by one, starting from the first function invoked. When performing a comparison against two functions, we utilized the Levenshtein distance to calculate the similarity of their function names. In our implementation, we added up all the similarity scores, and divided the sum by the total number of functions under comparison. This produced a normalized similarity score of the stack traces from the two bug reports. If the similarity of the stack traces is more than a threshold (*i.e.*, 0.8 used in ClusterFuzz), we treated them as duplicate bug reports. Otherwise, we deemed them as non-duplicated bug reports.

A.1 Klaus Appendix: Procedure of Manual Validation

In Section 4.6, it was stated that when a kernel error is encountered against an input (e.g., a PoC program), on a patched kernel, we revert the patch and rerun the input on the unpatched kernel. If the error occurs on the unpatched kernel and is different from the one observed on the patched kernel, it is referred to as manual validation. This process involves manually verifying if the error is a result of the patch. Below, we outline the steps involved in our human validation procedure.

When `KLAUS` generates a kernel error report, we first run the associated PoC program and capture a detailed record of the kernel execution using ftrace. If the kernel execution does not include the patch's code changes, we conclude that the patch is correct since the relevant code was not executed. In other cases, from the error report, we extract the series of kernel function calls leading up to the error, along with the relevant kernel object. Starting from the latter, we perform manual backward analysis to pinpoint the root cause of the error. This enables us to determine the correctness of the patch and conclude our human validation process.