UNIVERSITY OF CALIFORNIA
RIVERSIDE

N-day Vulnerabilities: Detection, Bisection, and Measurement

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Zheng Zhang

March 2025

Dissertation Committee:

Dr. Zhiyun Qian, Chairperson
Dr. Heng Yin
Dr. Manu Sridharan
Dr. Chengyu Song

The Dissertation of Zheng Zhang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to express my deepest gratitude to my advisor, Prof. Zhiyun Qian, for his unwavering guidance and support. He has always provided me with invaluable advice, ranging from the intricacies of research to practical methods of working. When I first met him, I knew very little about computer science. Under his patient mentorship, I gradually grew in knowledge and skill, eventually earning my Ph.D. in Computer Science. His passion for research and steadfast dedication to his career have deeply inspired me; he is one of my most important role models.

I am also very grateful to my committee members, Prof. Heng Yin, Prof. Chengyu Song, and Prof. Manu Sridharan, for their invaluable advice and insightful discussions throughout my dissertation.

I am deeply grateful to my lab mates and colleagues at UCR. Our discussions and collaborations have been both productive and enjoyable, and their support outside the lab has been invaluable throughout my Ph.D. journey. I would like to especially thank the following individuals (in no particular order): Hang Zhang, Dongdong She, Zhongjie Wang, Daimeng Wang, Yue Cao, Shitong Zhu, Yizhuo Zhai, Weiteng Chen, PengXiong Zhu, Yu Hao, Keyu Man, Xiaochen Zou, Xingyu Li, Xinan Zhou, Zhenchuan Liang, Haonan Li, Guoren Li, Juefei Pu, Qing Deng, Shenghan Zheng, Zhutian Liu, Yuan Tan, Dazhi Feng, Yifan Wu, and many others.

During my long years at UCR, I received help from many people, including both my professors and administrative staff. I regret that I cannot list everyone here, but your tremendous support was essential in helping me navigate this journey. I would like to give

special thanks to Vanda Yamaguchi. From course selection to internships and graduation, your patience and assistance were invaluable at every important step along the way.

Above all, I am most grateful to my family. No matter where I am or how far apart we are, my parents and grandparents have always supported me in every possible way. Whenever I faced challenges, I could always feel your unwavering support. A heartfelt thank you to my uncle and aunt—we have always shared a special connection. Though we made independent choices, fate brought us together across thousands of miles. Over these past few years, I have been fortunate to receive so much care from you. Lastly, I want to thank my little cousin. Watching you grow stronger and more outstanding has been a great source of encouragement for me.

To my family for all the support.

ABSTRACT OF THE DISSERTATION

N-day Vulnerabilities: Detection, Bisection, and Measurement

by

Zheng Zhang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2025
Dr. Zhiyun Qian, Chairperson

Open-source projects are widely reused in commercial software, yet its collaborative nature exposes it to significant security challenges, particularly N-day vulnerabilities. These vulnerabilities remain exploitable after patches have been released, largely due to delayed patch propagation in decentralized ecosystems. This research addresses the critical issue of prolonged vulnerability exposure by exploring the underlying causes of patch delays and developing automated tools that can help accelerate the patch porting process and reduce the window for attackers.

We first present a comprehensive measurement study of the Android kernel patch ecosystem, which systematically analyzes how security patches move from the Linux mainline through various layers of customization by chipset manufacturers and OEM vendors. Our findings indicate that patch delays are a systemic issue, with some patches taking months—or even over a year—to fully reach end-users, which increases the risk of exploitation. We analyzed the underlying causes, and one significant reason is that maintainers lack knowledge about which versions are affected by vulnerabilities. In other words, they

are unsure when a vulnerability was introduced and which versions are impacted, making it unclear whether the versions they maintain need to be patched.

Based on the above observations, we need to speed up the patch porting process to reduce the attack window of N-day vulnerabilities. Identifying the affected versions of these vulnerabilities is crucial for the patch porting process. Therefore, we tackle the challenge of bug bisection—the process of tracing vulnerabilities back to their originating commits. Traditional methods, such as dynamic testing and heuristic-based BIC (bug-inducing-commit, the change that first introduced the vulnerability into the codebase) identification, have shown limitations due to environmental inconsistencies and oversimplified assumptions. To overcome these issues, we introduce a novel approach that uses under-constrained symbolic execution to analyze code statically across multiple versions. This method precisely identifies whether the vulnerability logic exists in a given version, thereby isolating the bug-inducing commit.

However, the above method still faces several limitations. It requires a proof-of-concept, supports only a narrow range of bug types, and its accuracy is not very high (although it is higher than that of traditional methods). These shortcomings drive us to explore alternative approaches. Finally, we enhance bug bisection by employing large language models (LLMs) that combine code diffs and contextual commit messages. This multi-step filtering approach, which uses both coarse-grained and fine-grained analysis, significantly improves the accuracy of vulnerability detection. Together, these integrated techniques can help accelerate the patching process and reduce the exposure window for N-day vulnerabilities, contributing to a more secure open-source ecosystem. These contributions offer

practical solutions for swiftly mitigating vulnerabilities, enhancing open-source security, and ensuring robust resilience in critical software systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Open-source software is the foundation of countless critical systems and devices, yet its collaborative nature comes with inherent challenges in ensuring security [7, 96]. One of the most significant challenges is the phenomenon of N-day vulnerabilities—security flaws that remain exploitable well beyond the point at which they have been discovered and disclosed (in many cases, even after the initial patch is released). These vulnerabilities present a serious risk because attackers can take advantage of the lag between the discovery of a flaw and its complete remediation across all systems [88].

In many open-source ecosystems, software is developed in a highly decentralized fashion. Projects like the Linux kernel or Android operate under a model where multiple parties, ranging from upstream developers to various downstream vendors, contribute to the evolution of the code [96, 119, 14, 156]. For instance, the Android ecosystem is built upon the Android Open Source Project (AOSP), which itself is derived from the Linux kernel.

This code is then further customized by chipset manufacturers and original equipment manufacturers (OEMs) before finally reaching end-users. Such a multi-tiered process can introduce significant delays in the propagation of security patches. A patch that is applied in the upstream repository may take months to reach the final product, leaving users vulnerable during the interim.

To decrease attack windows, we need to accelerate the patching process of N-day vulnerabilities. To accelerate the patching process, we must quickly identify the scope of N-day vulnerabilities (for example, determining which stable Linux versions are impacted by vulnerabilities discovered in the Linux mainline) so that we can promptly notify affected downstream maintainers. To accurately identify the affected range, it is crucial to understand its origin and the precise point in the software's evolution when the flaw was introduced [58, 170, 194]. This task, known as bug bisection, is defined as tracing back through the commit history to identify the bug-inducing commit (BIC)—the change that first introduced the vulnerability into the codebase.

Traditional bug bisection methods have typically relied on dynamic testing approaches, such as executing a proof-of-concept (PoC) across multiple versions, or on heuristic-based static analyses of code diffs. However, each of these methods has notable limitations. Dynamic testing approaches, for example, are often hampered by inconsistencies in the execution environment. Variations in build configurations or runtime environments can lead to false negatives or false positives, making it difficult to reliably determine whether a particular version remains vulnerable. Additionally, executing a PoC across different versions may inadvertently trigger unrelated bugs, further complicating the analysis. On the other

hand, heuristic-based static methods generally assume a direct correlation between the code changes indicated in a patch and the bug-inducing commit. This assumption oversimplifies the complexity of software evolution. In many cases, the vulnerability is not solely localized to the lines of code modified by the patch; rather, it may result from a subtle interaction between multiple commits or from code that is logically related but not immediately adjacent in the commit history.

Overall, the background reveals two intertwined problems: the delayed propagation of security patches in complex open-source ecosystems and the difficulties in accurately identifying the commits responsible for introducing vulnerabilities. Both issues significantly contribute to the persistence and severity of N-day vulnerabilities. The inherent complexity of multi-party development environments and the limitations of traditional analysis techniques underscores the need for innovative approaches that can accurately identify affected versions of N-day vulnerabilities, thereby accelerating the patch porting process and ultimately reducing attack windows.

1.2 Overview

To address these challenges, a comprehensive research effort has been undertaken that spans the measurement of patch delays, the development of advanced bug bisection techniques, and the integration of novel approaches. The overarching goal of this work is to mitigate the risks associated with N-day vulnerabilities by both understanding the underlying causes of patch delays and by providing effective tools for accurate N-day vulnerability detection.

The first component of this research is an in-depth measurement study of the Android kernel patch ecosystem. By systematically analyzing the propagation of patches across multiple layers—from the initial commit in the Linux mainline to the final deployment by OEM vendors—this work provides quantitative insights into the timing and bottlenecks that hinder the swift deployment of security updates. The study examines a wide array of data sources, including public security bulletins, firmware releases, and commit logs, to build a detailed picture of the patch propagation process. The findings reveal that patch delays are a systemic issue, often extending to several months or even over a year, which significantly increases the window of vulnerability for billions of devices. Our analysis showed that one important reason behind this is that maintainers do not know which versions are affected by vulnerabilities. They are uncertain about when a vulnerability was introduced (In other words, they don't know which commit introduced the bug) and which versions are impacted, so they cannot easily determine if their maintained versions need a patch. This empirical evidence establishes a strong motivation for developing more effective solutions to accelerate patch propagation.

To accelerate the process of patch propagation, it's important to identify affected versions of N-day vulnerabilities. Thus the next phase of the research focuses on the problem of bug bisection—the process of identifying the specific commit that introduced a vulnerability. Given the limitations of traditional methods, we introduces a new approach called SymBisect [210] that leverages under-constrained symbolic execution. This technique involves statically analyzing the code across multiple versions to determine the presence or absence of the vulnerability logic. By symbolically executing relevant code paths, it becomes

possible to accurately assess whether a vulnerability exists in a particular version, thereby enabling the precise identification of the bug-inducing commit. The symbolic execution framework is carefully designed to focus on the portions of code that are most relevant to the vulnerability, thereby addressing the scalability issues (path explosion) that have traditionally limited the applicability of such techniques.

However, SymBisect still has several limitations, such as the requirement for a PoC, supporting only limited bug types (e.g., use-after-free and out-of-bounds memory access without race conditions), and relatively low accuracy (around 75%). These shortcomings motivate us to explore new approaches. Specifically, building on the advances in large language models (LLMs), the final component of this research introduces the use of LLMs to enhance the bug bisection process further (called LLMBisect). Modern LLMs have demonstrated impressive capabilities in understanding both programming languages and natural language, which opens up new possibilities for analyzing patches in their full context. Unlike conventional methods that rely solely on code diffs, the LLM-based approach takes into account the rich contextual information contained in commit messages as well. These messages often provide valuable insights into the intent behind code changes, the rationale for modifications, and hints about the underlying vulnerability logic. By incorporating this unstructured natural language information into the analysis, the research achieves a more nuanced understanding of the patch context. The LLM-based method is implemented through a multi-step filtering approach. Initially, a coarse-grained filtering stage rapidly narrows down the pool of candidate commits by utilizing simple patch information and keyword matching. This preliminary stage is computationally efficient and

significantly reduces the number of commits that require more intensive analysis. In the subsequent fine-grained filtering stage, the LLM is employed to conduct an in-depth analysis of each candidate commit. The model evaluates both the code changes and the accompanying commit messages to determine whether the candidate is likely to be the bug-inducing commit. To ensure robustness against inconsistencies and potential false positives inherent in LLM outputs, a majority voting mechanism is integrated at key stages of the analysis. This multi-step process has been shown to dramatically improve the accuracy of identifying bug-inducing commits, achieving performance levels that surpass traditional approaches.

Note that LLMBisect and SymBisect can complement each other. Although LLMBisect has several advantages, it requires a patch, whereas SymBisect can be used in cases where a PoC exists but a patch has not yet been developed (for example, new bugs exposed by fuzzers). By integrating these three components—empirical measurement of patch delays, symbolic execution-based bug bisection, and LLM-powered bug bisection—this research offers a holistic approach to mitigating N-day vulnerabilities. The measurement study provides a solid foundation by highlighting the urgency and scale of the patch propagation problem, while the advanced bisection techniques deliver practical tools for rapidly identifying the affected versions of vulnerabilities. The combined efforts aim to reduce the exposure window of N-day vulnerabilities and enable faster, more targeted remediation. In doing so, the work not only advances the state of the art in vulnerability detection but also offers actionable insights for improving the overall security posture of open-source software ecosystems.

1.3 Roadmap

The structure of this dissertation is as follows: Chapter 2 presents our measurement study of the Android patch ecosystem. Chapter 3 introduces SymBisect, a tool that automatically identifies bug-inducing commits in fuzzer-exposed vulnerabilities using under-constrained symbolic execution. Chapter 4 describes LLMBisect, which leverages large language models to enhance bug bisection by analyzing both code changes and contextual commit messages to precisely locate the bug-inducing commit. Chapter 5 concludes the dissertation.

Chapter 2

An Investigation of the Android Kernel Patch Ecosystem

Abstract

Open-source projects are often reused in commercial software. Android, a popular mobile operating system, is a great example that has fostered an ecosystem of open-source kernels. However, due to the largely decentralized and fragmented nature, patch propagation from the upstream through multiple layers to end devices can be severely delayed. In this paper, we undertake a thorough investigation of the patch propagation behaviors in the entire Android kernel ecosystem. By analyzing the CVEs and patches available since the inception of the Android security bulletin, as well as open-source upstream kernels (*e.g.,* Linux and AOSP) and hundreds of mostly binary OEM kernels (*e.g.,* by Samsung), we find that the delays of patches are largely due to the current patching practices and the lack of knowledge about which upstream commits being

security-critical. Unfortunately, we find that the gap between the first publicly available patch and its final application on end devices is often months and even years, leaving a large attack window for experienced hackers to exploit the unpatched vulnerabilities.

2.1 Introduction

Open-source software is ubiquitous and often serves as the foundation of our everyday computing needs. Unfortunately, they also contain a large number of vulnerabilities — there are new security patches released weekly for open-source software (e.g., Linux).

It can be tricky to ensure timely delivery of patches for open-source software because of the widespread *reuse* phenomenon where multiple versions or branches of the open-source software co-exist and can be divided into so-called *upstream* and *downstream* ones. Downstream developers *reuse* much of the upstream software and add finishing touches (*e.g.,* customization, stability fixes). More importantly, downstream developers have to take critical security patches from upstream to eliminate vulnerabilities. This is often challenging because upstream and downstream branches are often developed and maintained by different organizations and companies that often have different priorities and goals in mind.

The single most prominent example is the Android ecosystem. The Android open-source Project (AOSP) kernels are derived from Linux kernels (*i.e., reused* in Android) with many features added for mobile devices. In turn, the AOSP kernels are *reused* by chipset vendors such as Qualcomm who add additional hardware-specific changes. A chipset vendor's kernel is then finally *reused* by OEM vendors such as Samsung and Xiaomi. This

means that the patches can originate from more than one *upstream* kernels (e.g., Linux, AOSP, and Qualcomm), and the propagation can take multiple steps to finally reach the OEM vendors. Even though Google has been working diligently with OEM vendors on patching, e.g., through its monthly update program [7], the ecosystem is unfortunately so decentralized that it is beyond the control of a single entity.

Motivated by the lack of transparency and understanding of the patching process, we set out to investigate the unique and complex Android kernel ecosystem. Specifically, we are interested in the following high-level aspects:

(1) The relationship between the upstream and downstream kernels, *e.g.,* who is responsible for the initial patch, and how does it propagate?

(2) The timeliness of patch propagation, *e.g.,* what is the typical delay in each step with the patch propagation and where is the bottleneck?

(3) The factors that influence the patch propagation, *e.g.,* what are the current best practices by different entities, and how can we improve the situation?

It is challenging to conduct such a measurement study. Specifically, even though Android kernels inherit the open-source license from Linux, kernel sources from OEM vendors are often released broken/half-baked, with substantial delays, and only intermittently (*e.g.,* when the phone was initially released) [155, 149]. In contrast, the binary ROMs (*i.e.,* firmware images) are easier to find. Therefore, to be able to analyze closed-source Android firmware images, we build a static analysis tool on top of FIBER [207], a state-of-the-art tool capable of conducting patch presence test in binaries.

By analyzing the patches announced in the Android security bulletin, 20+ OEM

phone models, and 600+ kernel images, we delineate many interesting findings that reveal intriguing relationships among different parties as well as the bottleneck of the whole patch propagation process. When fair to do so, we also compare the responsiveness among different parties, *e.g.,* which OEM vendors are more diligent in patching their devices.

We summarize our contributions as follows:

- We investigate the unique Android kernel ecosystem that is decentralized and fragmented. We mine the patch propagation delays across all layers and locate the bottleneck.
- We improve a state-of-the-art source-to-binary patch presence test tool and develop a system on top of it to check the closed-source kernels from OEM vendors. We plan to open-source our system and release the dataset to improve the transparency of the ecosystem.
- We conduct a large-scale measurement that shows nearly half of the CVEs are patched on OEM devices roughly 200 days or more after the initial patch is publicly committed in the upstream, and 10% – 30% CVEs are patched after a year or more.
- Furthermore, by mining the commit methods and correlating them with notification dates published by Google and Qualcomm, we explain the causes of patch delays. We also distill takeaways and potential prescriptive solutions to improve the current situation.

2.2 Android Kernel Ecosystem

Android is known for its diverse and fragmented ecosystem where multiple variants of the operating system co-exist [96]. On one hand, the scale and diversity of the ecosystem

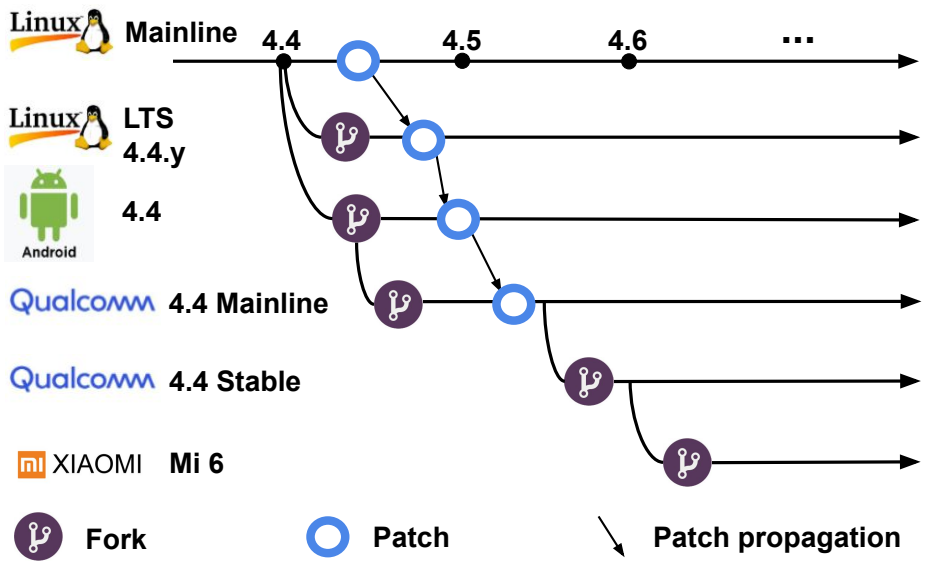

Figure 2.1: Android ecosystem for kernel version 4.4

participants definitely contributed to Android's overall success. On the other hand, it is extremely challenging to ensure the consistency and security of every Android variant out on the market. It is especially true for Android kernels which are themselves derived from the upstream Linux kernel.

Hierarchy of Linux/Android kernels. Figure 2.1 illustrates the typical relationship between the upstream and downstream kernels. At the very top, we have the Linux mainline that moves forward rapidly with all the features and bug fixes. Its kernel versions are tagged as 4.4, 4.5, *etc.* Periodically when appropriate, it gets forked into *stable* (*e.g.,* 4.3.y) or *long term support* (LTS) branches (*e.g.,* 4.4.y) with mostly only bug fixes [120]. The difference between *stable* and LTS branches is that the former is short-lived (a few weeks) while the latter is supported for a few years. For the benefit of longer support, Android common kernels (*e.g.,* 4.4) typically follow the LTS branches. Meanwhile, Google developers will add the necessary changes for mobile devices to turn the Linux kernel into an Android

kernel [119]. In addition, the developers will merge the fixes from Linux to ensure that they stay up-to-date and bug-free.

In Figure 2.1, Google's Android common 4.4 is initially forked from Linux mainline 4.4 and in the future merges all the changes from Linux LTS 4.4.y. Then there are branches maintained by SoC vendors such as Qualcomm, MediaTek, and Exynos (out of which only Qualcomm provides the complete history in git repos). Take Qualcomm as an example, when the company decides to ship a new SoC like Snapdragon 835, it may choose to fork a then-recent Android common 4.4.y. In fact, there exists a generic 4.4.y branch and multiple chipset-specific branches all maintained by Qualcomm (simplified in Figure 2.1). Interestingly, sometimes Qualcomm may choose to fork directly from upstream Linux (*e.g.,* 4.9.y) instead of Android common. Nevertheless, it will still merge significant changes from Android common later on. According to our analysis, SoC vendors typically take fixes and security patches from its direct upstream, Android common, instead of Linux. This practice is reasonable as Google has already done a significant amount of work for the SoC vendors such as patch compatibility tests for Android kernels. However, this also increases the patch propagation delay due to the extra hop.

Finally, at the very bottom of the hierarchy is the OEM vendor kernel. Depending on the device model and its chipset, *e.g.,* a Xiaomi phone using Snapdragon 835, the corresponding branch from the SoC vendor will be forked (Qualcomm's 4.4.y). The OEM vendor may then optionally add new features (*e.g.,* Samsung's kernel hardening [162]) or simply only port bug fixes from the upstream (for smaller OEM vendors). However, when it comes to security patches, OEM vendors tend to have a tighter connection with Google

who monthly updates its Android security bulletin since 2015. According to our knowledge, Google serves as the main point of contact notifying OEM vendors about various security vulnerabilities even though the original patch may come from other parties (*e.g.,* Linux or Qualcomm). From Sep 2017, Qualcomm has also established its own security bulletin and independently notifies its customers about Qualcomm-specific vulnerabilities [70, 156], which overlap with the ones on the Android security bulletin.

Android security bulletin is a central location where Google publishes monthly updates on Android security patches and their corresponding CVEs [7]. For the CVEs affecting the open-source Android components (for kernels, most are open-source except some proprietary drivers, *e.g.,* by MediaTek), there will be links to the upstream kernel commits representing the patches of the vulnerabilities.

It is worth noting that as Android kernels can be customized by individual OEM vendors, the bulletin may not cover OEM-specific vulnerabilities (*e.g.,* an OEM device may use a custom file system). Nevertheless, it represents Google's best effort to keep track of vulnerabilities that affect the Android common kernel, the upstream Linux kernel, and SOC vendors (primarily Qualcomm). In fact, each CVE has a corresponding link to its patch (*i.e.,* a git commit) that belongs to one of the three kernel repositories.

Before publicizing the vulnerabilities on the Android security bulletin, Google notifies OEM vendors at least one month earlier to ensure that affected devices are patched [8]. In other words, the publication of vulnerabilities on the Android security bulletin represents a major event in the patch management cycle, after which unpatched devices will be in danger. Indeed, our measurement results suggest that OEM vendors are

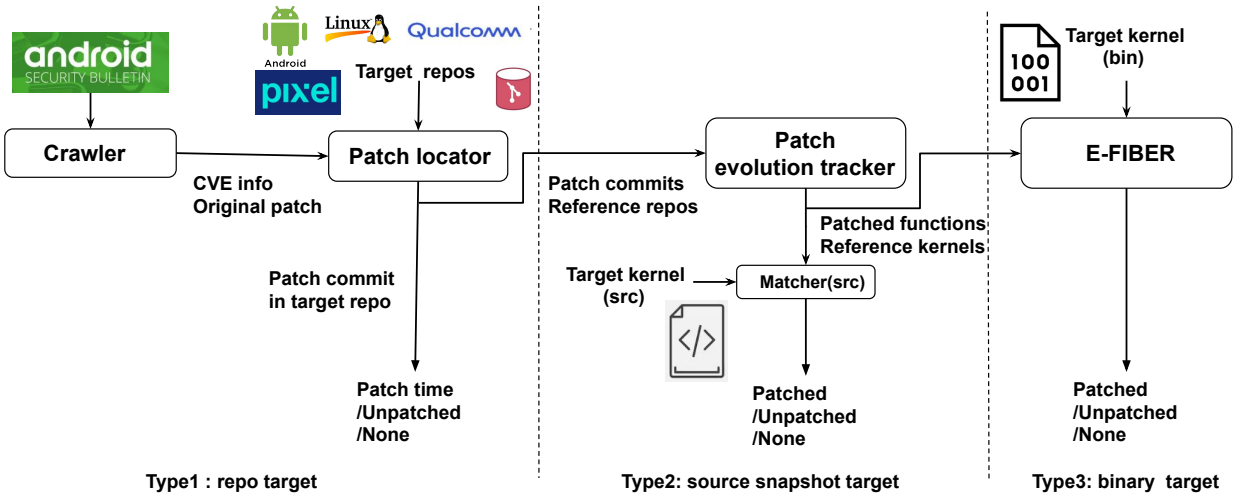

Figure 2.2: Measurement pipeline.

dependent on Google for patching.

2.3 Measurement Goal and Pipeline

As alluded to earlier, the goal of the measurement is to shed light on the patch propagation in the fragmented Android kernel ecosystem. In this paper, we explicitly assume the knowledge of the affected function(s) and the source-level patch itself, as the upstream Linux/Android kernels do offer detailed patch commits. As a result, our goal is that *given a CVE, we will track the propagation of the initial patch along the chain of upstream-downstream kernels.* Together with the CVE publication time on the Android security bulletin, we can paint a timeline of patch commit and announcement events in the whole patch management cycle.

Before we introduce the measurement pipeline, we first introduce the **three different types of kernels** that are publicly accessible, with increasing degrees of

difficulties to analyze.

(1) Type 1: Repository. Kernels made available through git repositories contain complete commit history. They represent the easiest case to analyze as a security patch can be easily located in the commit log — typically they simply copy the commit message and/or reference the commit given in the Android security bulletin's link. *Linux, Android common, Qualcomm and Nexus/Pixel* kernels belong to this category. Unfortunately, other SoC vendors such as Samsung Exynos, MediaTek, and Huawei Kirin do not offer git repositories corresponding their recent chipsets.

(2) Type 2: Source code snapshots. Most OEM vendors prefer to release their kernels in the form of source code snapshots without commit history (Google's own Nexus/Pixel phones are exceptions). It is usually possible to check if a particular CVE is patched in the snapshot via simple source-level function comparison (more details in §2.4.2). The issue though, is that such snapshots are released with substantial delays and often sporadically, leading to missing data points and inconclusive results.

(3) Type 3: Binary. The most available form of OEM kernels is the binary one – firmware images or ROMs. In fact, there is an abundant supply of Android ROMs on both first-party [39, 40] and third-party websites [36, 37]. These ROMs represent a valuable data source for patch propagation analysis, as long as we can accurately test patch presence in these binaries.

Measurement pipeline. Now we introduce the measurement pipeline (Figure 2.2) that integrates the analysis of the above three kernel types:

(1) Crawler. Initially, we crawl the kernel-related CVE information from Google's

Android security bulletin [7]. This includes CVE numbers, specific patch commits, and the corresponding repositories in which the patches were committed.

(2) Patch locator. This is to analyze type 1 target kernels (*i.e.,* repositories). It attempts to determine if a given patch (or a similar one) exists in a target kernel repository (§2.4.1). If so, it outputs the corresponding patch commit in the repository, which then also serves as the reference in the patch presence test for type 2 and type 3 kernels.

(3) Patch evolution tracker. The tracker tries to collect all possible versions of a patched function (*i.e.,* the function can continue evolving after the security patch) in the repositories, this can help us reliably test the patch presence in both type 2 (*i.e.,* source snapshot) and type 3 (*i.e.,* binary) kernels.

(4) Source-level matcher. It tries to match each patched function version (identified by the evolution tracker) to the target function in a type 2 kernel, in order to perform a source-level patch presence test (§2.4.2).

(5) E-FIBER. E-FIBER is capable of translating each patched function version into a binary signature and then matching the signature in type 3 binary kernel as a patch presence test. We build E-FIBER on top of FIBER [207], a state-of-the-art binary patch presence test system. We will articulate the improvements we made over FIBER in §2.4.3.

2.4 Patch Presence Test

In this section we will detail the methodology of patch presence tests against the three kernel types.

To better facilitate the discussion of this paper, we call the patch linked in Android security bulletin the "linked upstream patch", which can only be in type 1

kernels (repositories), *i.e.,* Linux, Android commons, Qualcomm. Interestingly, later we find that these may not be the earliest patches.

2.4.1 Repository Target

When our target is a repository, we search through the commit history using the patch locator to test the presence of an equivalent patch.

Patch locator: We combine various information about the original patch to determine its presence in the target repository. Specifically, we have the following procedure:

1) For each commit, we attempt to perform a simple string match on the commit subject. If it is a patch they borrow from the upstream, the downstream kernels typically retain the original subject. If there are multiple hits, we use the commit message to identify the real match. Typically, the downstream kernels will not only copy the original commit message but also reference the upstream commit, *e.g.,* `cherry picked from commit XYZ`. If no results are found, we perform the second step.

2) When commit subject and message are not retained when applying the same patch in downstream, we search through the commit history of the corresponding patched file, attempting to match the complete source level changes (including both the added/removed lines as well as the context lines) with those in the original patch. If still no match, we move to the next step.

3) It is possible that the downstream kernel has customized the patched function and its context lines no longer match those in the original patch. We therefore also attempt to match the added and deleted lines only (ignoring the context lines). However, if still no results are found, we keep the commits that matched with at least some blocks of added

lines (which we call "change sites") in the original patch.

In any of the above steps, if there are multiple results returned, we manually identify the correct one by inspecting the commit message (note that the message is no longer exactly copied else the first step would have caught it). In addition, if no match is ever found after all the steps, we attempt a manual search using parts of the message of the original commit as a last resort. Only if this step fails to locate any commit will we determine the commit is missing. In practice, we find these cases that require manual analysis are small (6.8% in our experiments).

In addition, there are several special cases we need to pay attention to:

(1) File path/name change: If we cannot find any commits that change the patched file, we extend the search region to files that have the same name but in different directories (sometimes the downstream kernel would decide to rearrange certain source files). If we find any commit that renamed the patched file at some points, we also track the evolution of the renamed file.

(2) Function name change: similar to file names, the name of a function may also change over time. We develop a small script to track the evolution of them too by checking the related commits.

(3) Patched at initialization time: sometimes a kernel repository or branch may choose to copy the entirety of a source file and commit it as a brand new file. In that case, we lose the actual commit that applied the patch. However, we can still match the change sites given in the original patch.

Finally, we note that there can be several reasons when a patch is not found: 1)

the patched file/function simply doesn't exist in this branch (*e.g.,* a vulnerable Qualcomm driver is not used in Huawei devices), 2) the vulnerability does not affect the particular branch/repository, 3) The vulnerability fails to be patched. In our evaluation, we consider a CVE not applicable for a particular target if it falls under case 1).

2.4.2 Source Code Target

For kernel source snapshots, we need a way to check its source code against the patched version and infer the patch presence. A naive approach is to match the patched function from upstream against the same function in the snapshot. However, there can be multiple versions of the patched functions (*i.e.,* due to further commits to the same functions), and we do not know which version the target may take (regardless of whether it is source code or binary target). Even worse, the patched function name or patched file may change altogether as mentioned previously.

Our solution to this problem is straightforward. In addition to the single version of a patched function, we choose *multiple versions of the patched function* to represent the patch of a vulnerability. In general, we have two criteria to select the versions we should consider:

(1) Complete. We should be able to discover all patched versions of a function — unless the version is internal to the OEM and not visible in the upstream kernel repositories due to vendor-customization.

(2) Unique. The patched version should not occur in the unpatched version of the kernel. Otherwise, it no longer can distinguish the patched and unpatched cases.

Patch evolution tracker: In order to generate a complete set of patched function

versions, we need to pick one or more reference kernels first where we can track the evolution of a function post-patch — this means that we must use kernel repositories with commit history as reference kernels.

In this paper, we choose the repositories from Qualcomm as our reference kernels. This is because Qualcomm has the largest market share as a chipset vendor and therefore is the direct upstream of most Android devices. If a bug is fixed in Linux or Android common kernels, they should also exist in Qualcomm; in other words, Qualcomm has a superset of patches.

Qualcomm maintains different repositories for several major kernel versions (*e.g.,* 4.4 and 4.9). Within each repo, there is typically a "general release branch" (which we simply refer to as mainline) and multiple "stabilization branches" (which we refer to as stable) exist [68]. A stable branch usually corresponds to specific chipsets and OS versions (*e.g.,* Android 8.0) and only port fixes from the mainline. For example, branch `kernel.lnx.4.4.r34-rel` in repo `msm-4.4` has tags sharing a prefix of `LA.UM.7.2.r1` which corresponds to snapdragon 660 and Android 9.0 [69].

As any OEM kernel either forks from or follows a corresponding Qualcomm stable branch (which determines the chipset) and Qualcomm repo (which determines the kernel version), we choose the reference repo according to its kernel version. In practice, this minimizes the differences between the two and improves the accuracy of the patch presence test.

After choosing repositories, we need to determine in which branches to track the patched functions. In principle, we could choose all the branches (including mainline and

stabilization) but it may be unnecessary and time-consuming. Instead, we choose the mainline branch only for the following reasons: 1) Generally, vulnerabilities are patched in the mainline first and then propagated to the chipset-specific branches. Due to delays, the patch may not even exist in a chipset-specific branch but we cannot rule the vulnerability out. 2) We prefer to generate generic signatures which are not overly-specific; otherwise there may be too many signatures to generate in the end. In §3.6, we will show that this strategy produces satisfactory accuracy.

Source-level matcher After collecting the different versions of the patched functions in the corresponding repository, *e.g.,* Qualcomm 4.4, we need to compare them against the function in the target kernel. There are several ways to do so, *e.g.,* hash-based methods [59], a straightforward string match of a few representative lines (*e.g.,* changes made in the patch) in the function, or even a simple string match of the whole function.

We decide to use the most strict and simplest method — strict string matching of the whole patched function (using all the evolved versions post-patch) after stripped trailing white spaces for the following reasons: 1) It is strict and never produces any false positive, *i.e.,* if we claim that a function is patched, it must match some version of the patched function (and not any unpatched version). 2) The method is simple and easy to reason about. While it does produce false negatives, *e.g.,* the target kernel may customize the patched function so that it looks different but still patched, we find that these cases are uncommon and we are able to manually analyze them (given that we have the target kernel source).

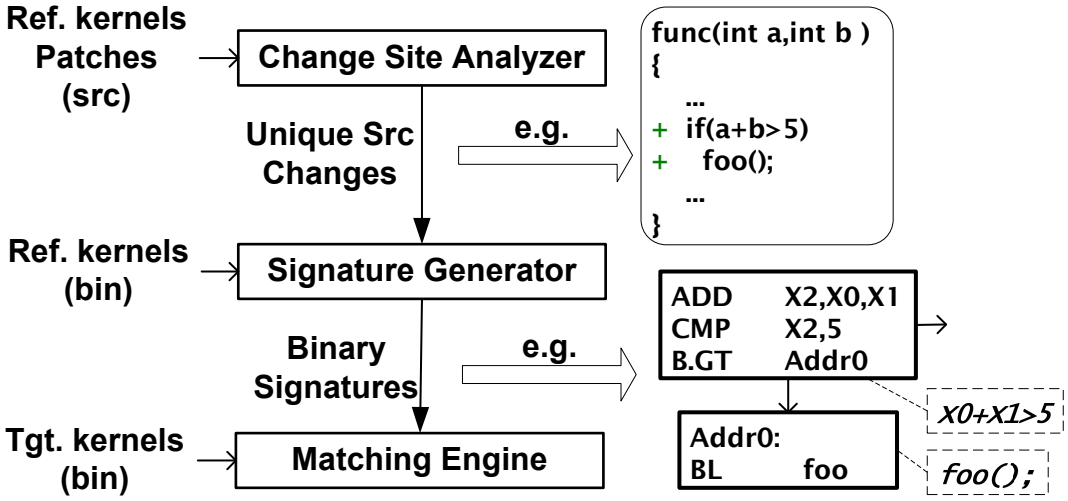

Figure 2.3: Fiber Workflow

2.4.3 Binary Target

If the target is a binary, neither of the previous two methods works. The key challenge is that the patched functions at the binary level are unlikely to be identical even if their sources are the same. This is because of various kernel and compiler options that can influence the compiled binary instructions. Therefore, we choose to generate binary signatures (in the patched function) to test the presence of patch in the target. The signature is what represents the semantics of a patch.

Specifically, we build an improved version of FIBER whose original workflow is illustrated in Figure 2.3. There are three main steps: 1) it first analyzes a patch (*i.e.*, changes made in one or more places) and checks the uniqueness of each change site. Then it picks a few suitable change sites for signature generation. 2) FIBER compiles the kernel and extracts relevant sequences of instructions (and even symbolic formulas involving the

computation of variables) representing the semantics of these change sites. 3) FIBER matches the signatures against a target binary.

Unfortunately, there are several limitations acknowledged and summarized in the original paper: 1) Function inline. (2) Function prototype change (3) Code customization. (4) Patch adaptation. (5) Other engineering issues. We observe that several of these issues share a common root cause: *patched functions evolve over time* and FIBER picks only the initial version of the patched function for signature generation. This means that if the release date of the target kernel and the original patch differ significantly, the generated signature is likely out-of-date for the target kernel. In our preliminary evaluation of FIBER spanning 3 years of reference and target kernels, we find that its accuracy dropped considerably compared to what was reported in [207] due to this issue.

To overcome this limitation, we simply leverage the patch evolution tracker (proposed earlier) to identify the multiple versions of the patched functions so that a more complete set of signatures can be generated. This is especially important when the change sites of the original patch are completely erased during the evolution of the patched function.

In addition, we also address two other technical problems mentioned earlier: (1) the patched function becomes inlined, and (2) the binary signatures look different for the same source due to different compilers and configuration options (FIBER has some degree of robustness but can still be affected as discovered in our preliminary analysis).

Function inlining can cause a direct failure in locating the patched function in the reference binary (missing from the symbol table) and therefore failure in generating the

Type of target	Company	Repo (Num of branches) or Phone models (Num of Roms)
Repository	Linux	Linux(mainline, linux-3.18.y, linux-4.4.y, linux-4.4.y, linux-4.14.y)
	AOSP common	Android common(android-3.18, android-4.4, android-4.9, android-4.14)
	Qualcomm	msm-3.18(8), msm-4.4(17), msm-4.9(15), msm-4.14(1)
	Pixel	Android msm (Pixel l, Pixel 2, Pixel 3)
Binary	Samsung	Galaxy S7(78), Galaxy S8(52), Galaxy S9(32), Galaxy Note9(28), Galaxy A9 Star(11), Galaxy A8s(9)
	Xiaomi	Mi 6(84), Mi8 Lite(24), Mi 8(12), Redmi 4(41), Redmi 4pro(38), Redmi Note7(21), Mi Max2(75)
	Huawei	Mate 10(37), P20 pro(31), Honor10(30)
	Oppo	R11s(11)
	LG	V30(10)
	Oneplus	Oneplus5(27), Oneplus6(18)
Source snapshot	Sony	XperiaXZ1(23)
	Samsung	Galaxy S8(1), Galaxy S9(1)
	Xiaomi	Mi 8(1), Mi 9(1), Mi Max2(1), Redmi Note7(1)
	Huawei	Mate 10(1), P20 pro(1)
	Oppo	FindX(1)

Table 2.1: Data set of measurements

signature.

Our solution is as follows: we try to find the caller of the patched function which should contain the inlined version of the patched function. If the caller is also inlined, then we will recursively locate the caller of the caller until one is found in the symbol table. Since the reference kernels are compiled by E-FIBER, we can make use of debug information to locate the exact sequence of instructions that belongs to the patched function (which is inlined), and generate the signatures (which are now in the context of a caller) accordingly. This signature can then be matched in the target kernel which has the same inlined behavior.

To address the compiler and configuration issues. We vary these configurations ahead of time in generating the binary signature.

(1) Compilers. Most vendors use GCC to compile their source code, however, a few new devices released in 2019 (whose corresponding Linux versions are 4.14) use Clang. Different compilers can yield vastly different binary instruction sequences to the point it

	repository	Num. CVEs
1	Linux	141
2	Qualcomm msm-3.4	12
3	Qualcomm msm-3.10	52
4	Qualcomm msm-3.18	115
5	Qualcomm msm-4.4	63
6	Qualcomm msm-4.9	15
7	AOSP msm	2

Table 2.2: Corresponding repository of CVE in Android security bulletin

becomes hard to semantically test the equivalence of the two. As a result, we use both compilers to compile 4.14 reference kernels and generate two versions of signatures.

(2) Optimization levels. Through sampling a few kernel source snapshots from major OEM vendors, we find that all of them use either Os or O2 as the compiler optimization levels. We, therefore, generate signatures with both optimization levels.

(3) Configuration files. Besides optimization levels, other kernel configuration options (to enable and disable certain kernel components) vary. In the mainline branch of Qualcomm repos (*e.g.,* 4.4 or 4.9), there are typically a few config files. For example, msm-4.9 has 16 config files in total and only 8 of them are specific to Android chipsets, including `sdm845-perf_defconfig` (Snapdragon 845), `msm8937-perf_defconfig` (Snapdragon 430), etc. We pick only the config files that are relevant to the Android devices we are interested in testing. For example, snapdragon 845 is used in Mi 8. Thus `sdm845-perf_defconfig` is used to generate the corresponding signatures.

2.5 Evaluation

2.5.1 Dataset

Overall, we collected 402 kernel CVEs released on Android Security Bulletin every month since its inception in Aug 2015 until May 2019. This includes the main bulletin [7] as well as a Pixel bulletin [21]. We summarize the crawled CVEs in Table 2.2. Clearly, most of them link to Linux and Qualcomm instead of AOSP Android repositories.

We also summarize the target kernels used in our evaluation in Table 2.1. Overall, we collected 3 levels of upstream kernels as introduced before, *i.e.,* Linux, Android common and Qualcomm. 8 most popular Android brands (Google Pixel, Samsung, Xiaomi, Huawei, Oppo, OnePlus, Sony, LG), covering 26 phone models and 701 released kernel instances (either source or binary). For most phone models, the kernel instances cover a time range of one to two years. We collect these kernels through both official and third-party websites. Our experience is that most official websites supply only the latest ROM for each phone model, and occasional source snapshots. The one exception is that SONY offers all source code snapshots on its websites. To obtain historical versions of ROMs, we rely mostly on third-party websites [41, 45, 36, 37].

We extract compilation dates (*i.e.,* build dates) from these ROMs which are used to compare against various dates such as Android security bulletin release date and patch dates on the upstream. Note that we collect many historical kernel versions (*e.g.,* 78 versions for Samsung Galaxy S7) for the same phone model in order to conduct a longitudinal study on their patching behavior.

To generate robust signatures using E-FIBER (see §2.4.2 and §2.4.3), we have

Device	Kernel Version	Source code						Binary					
		Cnt.	TP	TN	FP	FN	Accuracy	Cnt	TP	TN	FP	FN	Accuracy
Samsung S8	4.4.78	351	257	59	0	35	90.03%	246	202	37	0	7	97.15%
Samsung S9	4.9.112	302	293	3	0	6	98.01%	189	180	2	0	7	96.30%
Xiaomi Mi8	4.9.65	232	208	23	0	1	99.57%	168	149	15	0	4	97.62%
Xiaomi Mi9	4.14.83	262	258	3	0	1	99.62%	173	165	1	0	7	95.95%
Redmi Note7	4.4.153	356	342	13	0	1	99.72%	265	255	7	0	3	98.87%
Xiaomi Max2	3.18.31	328	217	88	0	23	92.98%	208	155	45	2	6	96.15%
Huawei P20	4.9.97	137	114	12	0	11	91.97%	83	76	5	0	2	97.59%
Huawei Mate10	4.4.23	147	74	67	0	6	95.92%	86	53	26	2	5	91.86%
Oppo FindX	4.9.65	235	210	19	0	6	97.45%	186	171	12	0	3	98.39%

Table 2.3: Accuracy of patch presence test

used in total 19 different config files from msm-3.18, msm-4.4, msm-4.9, and msm-4.14 Qualcomm repos that represent the chipsets encountered in our OEM devices. We use two compiler optimization settings: -Os and -O2. We also need to account for patch evolution. In the end, we compiled a total of 2,488 reference kernels all from Qualcomm repos with 11,093 signatures generated in the end (note one compilation allows multiple signatures to be generated).

2.5.2 Accuracy

In this section, we will describe the accuracy of patch presence test against three types of kernel targets presented in §2.4.

First of all, for kernels that are in the repository form, since we have conducted both automated and manual analysis (for the few subtle cases) exhaustively on every CVE and every branch, we treat the results as ground truth.

For kernels that are in source snapshots or binary ROMs, we sample a number of them to evaluate the accuracy of the patch presence test at both the source and binary level. Specifically, we picked 9 kernels, each from a different phone model covering 4 different

brands. These 9 kernels are available in both source snapshot and binary, which allows us to verify the results of binary patch presence test using the corresponding source code. The results are summarized in Table 2.3. Generally, our solution works well for both source and binary targets with an average accuracy of more than 96%. To give more details, we also analyzed the sources of inaccuracies.

In the case of source snapshot targets, since we consider a function patched only when a strict string match of the full function is found, it leads to no false positives but some false negatives are observed, which are due to customization of the patched functions. The results suggest that Huawei and Samsung have more customization than others. This is consistent with the fact that Samsung and Huawei are the top 2 players in the Android market and have the strongest product differentiation.

In the case of binary targets, the inaccuracies come from 1) Customization of the patched function. 2) Even when source code is the same, the binaries may look different due to vendor customization of compiler's config options, which we do not have complete access to (other than those from the periodic source snapshots). Interestingly, we can see generally comparable and even lower false negative rates compared to the source snapshot targets. This is because the source-level patch presence test is based on strict string matching of the whole patched function (and will fail to match any vendor customized functions). On the other hand, FIBER by design has some resistance against customization as the generated signatures only characterize a small (but key) portion of the patched function.

Besides, the number of CVEs and their corresponding patches that we can track for binary kernel targets is smaller. One common reason is that many vulnerable drivers

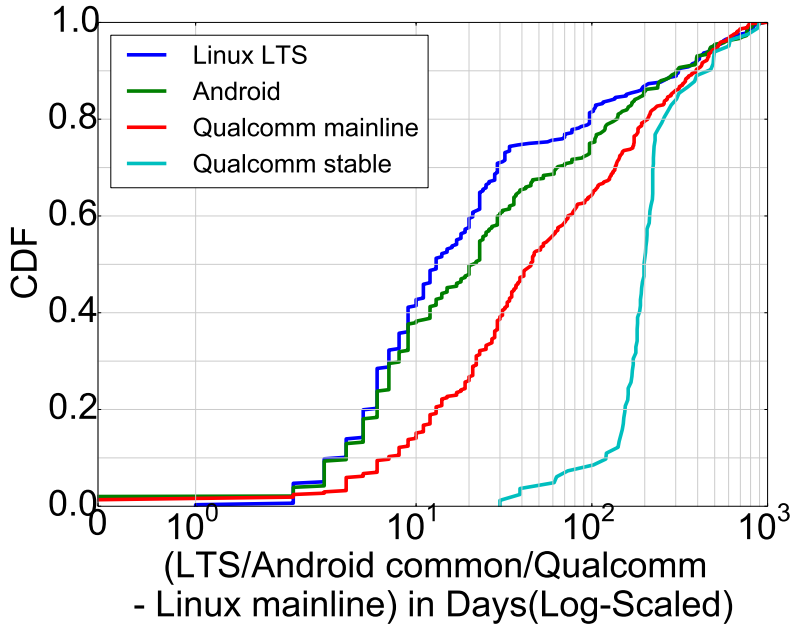

Figure 2.4: Upstream patch delays (Linux CVEs)

are included in the source snapshot but are not compiled into the binaries. Other technical reasons are: 1) FIBER was not able to generate signatures for certain cases. 2) Generation/Matching of signatures costs too much time (over a threshold of 2 hours, which is determined by the distribution of time consuming we observed). These cases attribute to about 10% of the CVEs and were excluded from the binary patch presence test.

Overall, the patch presence test accuracy result gives us confidence in the measurement study in §2.5.4. We also note that patch presence test in upstream source

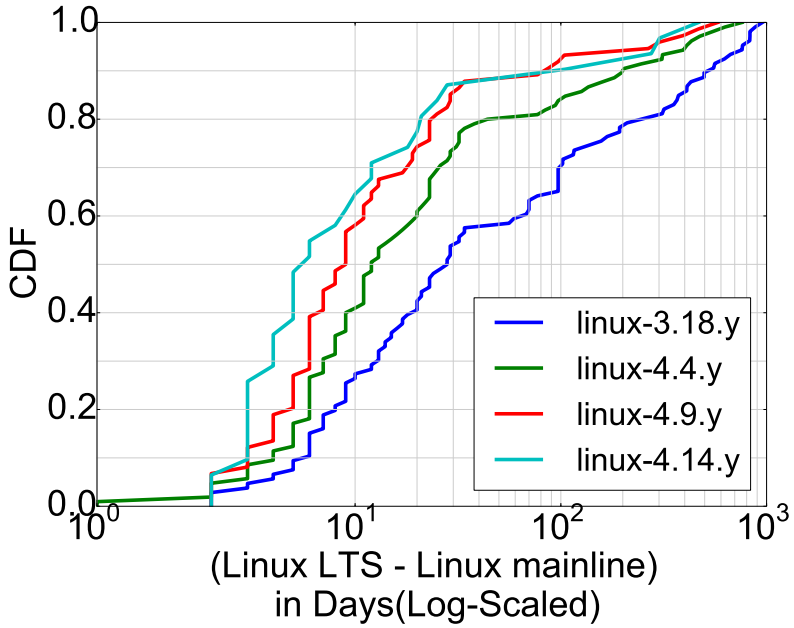

Figure 2.5: Linux mainline to LTS (Linux CVEs)

repos is independently done through patch locator as described in §2.4.1.

2.5.3 Patch Propagation in Upstream kernels

In this section, we focus on analyzing the patch propagation in the upstream kernel repos using the patch locator described in §2.4.1. With the exact time and date of individual commits, we are able to track the patch propagation precisely and make a number of interesting observations about both Linux and Qualcomm vulnerabilities.

Figure 2.4 gives an overview of the cumulative patch delays observed at each layer with respect to Linux mainline (here all included CVEs affect Linux). As we can see, Linux internally (mainline $\rightarrow$ LTS) already has a substantial delay, with 20% of the patches being 100 days or longer. On the other hand, Google does a good job in tracking

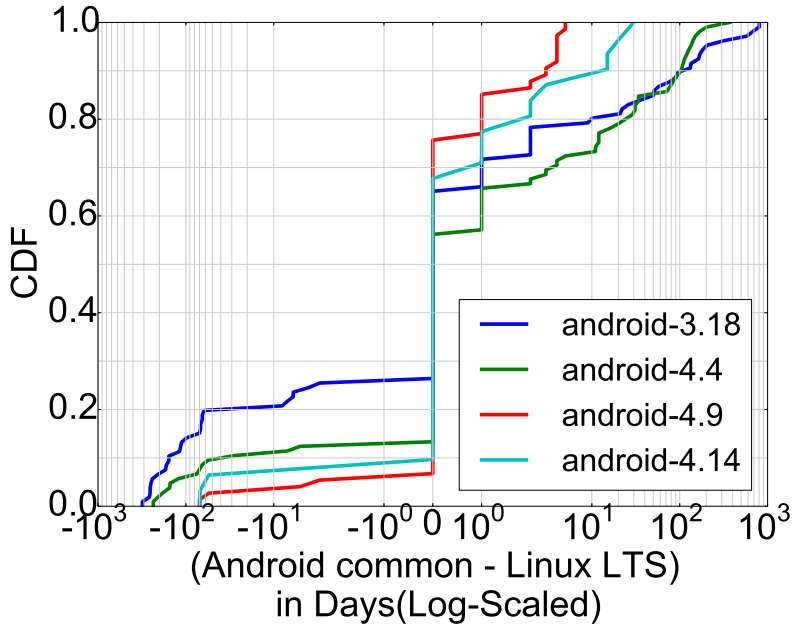

Figure 2.6: LTS to Android common (Linux CVEs)

Linux vulnerabilities, as the line representing the Android common's patch delays is closely aligned with that of Linux LTS. Qualcomm's mainline is noticeably slower in picking up patches from its upstream (note the log-scale nature of the X-axis). Finally, we find that Qualcomm can be considered the bottleneck as it is extremely slow in propagating most of its patches from mainline to stable branches. For about half of the cases, the Qualcomm-internal propagation delay is at least 2 to 3 months. From the end-to-end point of view, the majority of patches take over 100 days for them to propagate from Linux mainline all the way to Qualcomm stable. About 15% of the patches took 300 or more.

If we break the result down further layer by layer, Figure 2.5 shows the delay incurred in Linux internally (mainline $\rightarrow$ LTS) across all four major kernel versions 3.18,

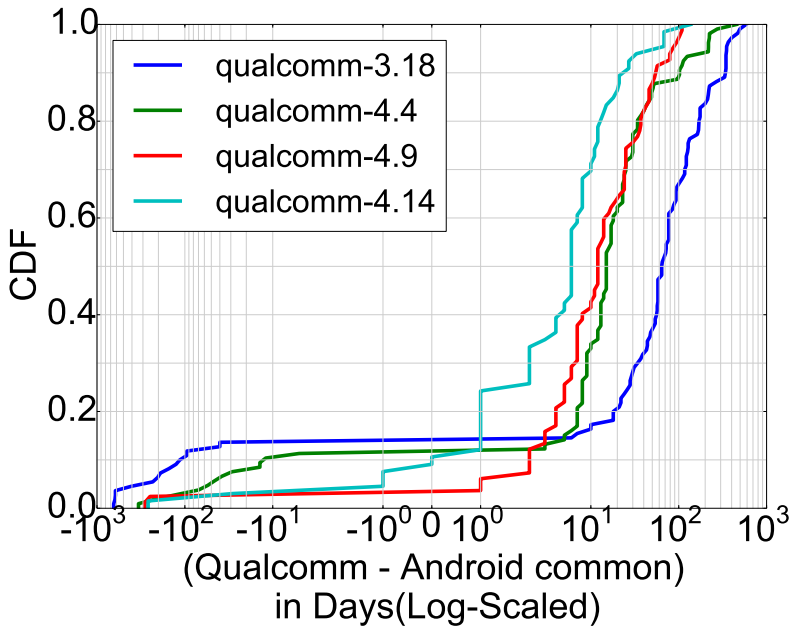

Figure 2.7: Android to Qualcomm mainline (Linux CVEs)

4.4, 4.9 and 4.14. We see 5% to 25% of patches experience a delay of 100 days or longer (with 3.18 being the worst). In extreme cases, after patched in Linux mainline, CVE-2017-15868 is not patched in Linux LTS 3.18 until 954 days later. Not too long ago, a critical vulnerability CVE-2019-2215 was not patched in Linux LTS 4.4 until about 600 days later, ultimately leaving most downstream OEM kernels such as Pixel2 and Samsung S8/S9 vulnerable [102].

The case for Linux LTS → Android common (Figure 2.6) is different and interesting. The delays are much smaller where more than half of the CVEs are patched in Android common the same day as Linux LTS or earlier. When we look into the reason, we find that the maintainer of Linux LTS, Greg Kroah-Hartman, also helps maintain the Android common repository (note the large fraction of 0-day delay cases). After merging

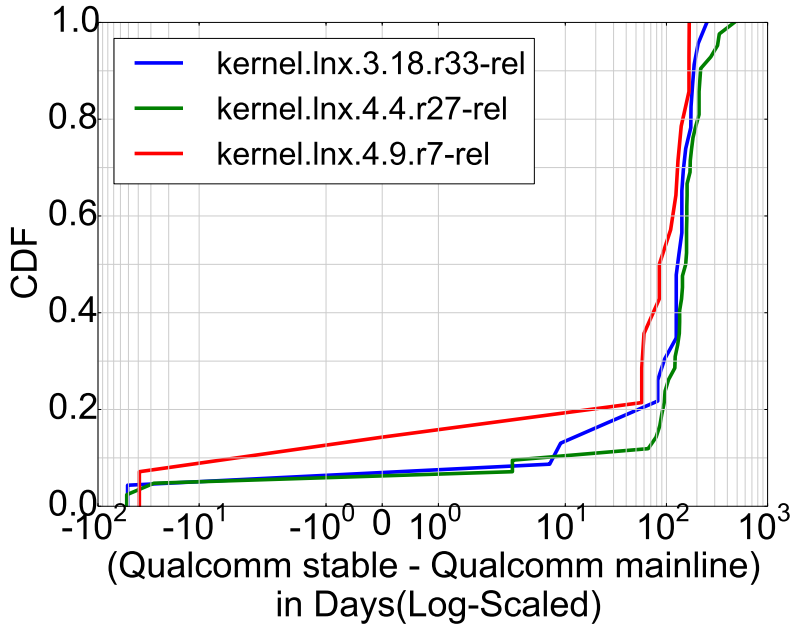

Figure 2.8: Qualcomm mainline to stable (Linux CVEs)

commits from mainline to LTS, he usually merges commits from LTS to Android common repository right away. The other thing worth noting is that about 10% – 20% of the patches are applied in Android common first and then appear in LTS, exhibiting negative delays. This is because Google has been diligently scouting for important security patches everywhere, sometimes picking up patches from Linux mainline directly and bypassing the slow Linux LTS. Google is capable of doing this because (1) they hire many engineers who are also Linux maintainers, and (2) Google offers a bug bounty program and thus many Linux bugs are reported to Google first who typically tries to get Linux mainline to patch first and then port it immediately (according to the feedback we received from Google).

The case for Android common → Qualcomm mainline (shown in Figure 2.7) is

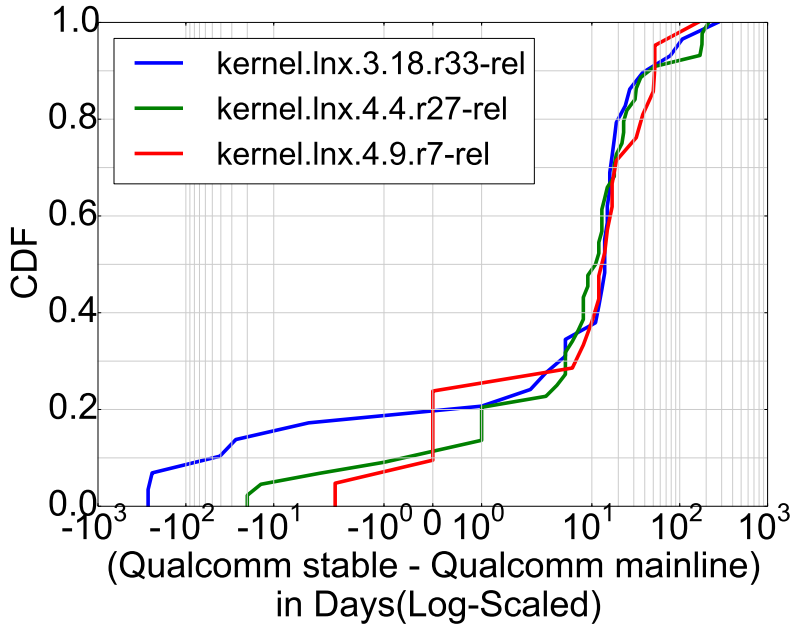

Figure 2.9: Qualcomm mainline to stable (Qualcomm CVEs)

similar in the sense that also about 5% – 20% of the patches are observed in Qualcomm first and then Android common. Similar to Google, Qualcomm also independently ports patches from Linux mainline. Interestingly, this means that even after Google picked up patches from Linux mainline directly, there are additional mainline patches missed by Google which are picked up by Qualcomm directly.

The last step in the pipeline is about the Qualcomm mainline branch (e.g., 3.18) to its corresponding stable. As shown in Figure 2.8, we pick three representative stable branches that correspond to the Android devices and OS versions we will analyze (recall that stable branches are specific to chipsets and Android OS versions). We note that other branches yield similar results (except those ones with insufficient history). We excluded all

4.14 stable branches because they are too new to have sufficient history. Overall, we can see that the delay is very substantial compared to the earlier steps. For 4.4, about 80% of the patches are delayed for 100 days or longer and 20% delayed for 200 days or longer. 4.9 is somewhat better than 4.4 with 80% of the patches delayed for 60 days or longer. Both are far worse than the internal delays in Linux (Figure 2.5). Interestingly, the 3.18 stable branch shows a comparable delay to 4.4 (and even slightly better) — a sharp contrast with the previous step that the Qualcomm 3.18 mainline being the slowest among all other mainlines (shown in Figure 2.7). Upon closer inspection, this is due to an older patching practice for the Qualcomm 3.18 repo which we will discuss in detail in §2.6.

In summary, for vulnerabilities that originate in Linux, we pinpoint the *internal propagation delays* within Qualcomm and Linux (*i.e.,* mainline to stable/LTS) to be clear bottlenecks. In addition, we find that newer kernel versions (from 3.18 to 4.14) generally correspond to more timely patch propagation across all these layers. The improvement however appears to have stabilized since 4.9.

Finally, we also inspect vulnerabilities that originate in Qualcomm — they constitute more than 60% of the CVEs as shown in Table 2.2. Surprisingly, as shown in Figure 2.9, the patch delays seem abnormally small compared to the Linux vulnerabilities (Figure 2.8). We suspect this is because Qualcomm is much more aware of the vulnerabilities specific to its own code, *i.e.,* triaged and analyzed internally, and thus can react faster. We will provide more evidence to support this in §2.6.

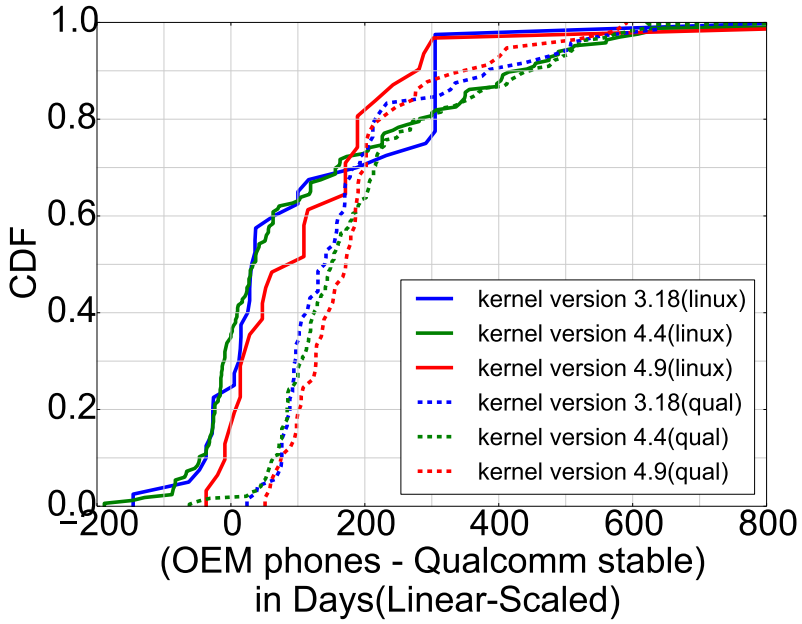

Figure 2.10: Delay between Qualcomm stable and OEM phones

2.5.4 Patch propagation to Android OEM phones

In this section, we follow the patch propagation pipeline to OEM vendors using a variety of Android devices as described in §2.5.1. We are primarily interested in measuring the patch delay and understanding generally whether OEM delays represent the bottleneck in the end-to-end patch propagation. In addition, these Android devices are produced and maintained by different companies, marketed as high-end or low-end phones, and released in diverse geographic regions. We therefore also examine how these factors may influence the patching behavior. For most phones, we are able to retrieve a continuous stream of firmware images (one image per month according to build dates). Thus we can pinpoint when a patch is applied.

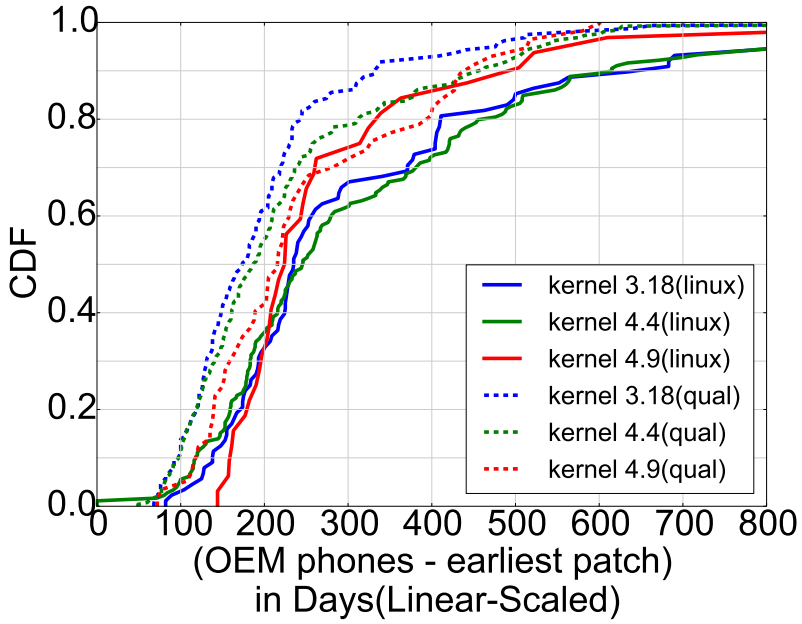

Figure 2.11: End-to-end delay between earliest patch and OEM phones

Figure 2.10 shows the patch propagation delay from Qualcomm stable to OEM phones (aggregated over all the phones). For every OEM phone, we pick one or more corresponding Qualcomm stable branches as upstream with the matching chipset and Android OS versions (note a phone may upgrade its Android OS version during its lifetime). As we can see, for Qualcomm-specific vulnerabilities (in dotted lines), OEM phones fall behind Qualcomm stable significantly — the delay is 100 days or more for 70 - 90% of CVEs. On the other hand, for vulnerabilities that originated in Linux, we find that the delays are noticeably smaller. This is due to Linux vulnerabilities being patched much earlier in upstream (Linux and Google's Android common) and therefore OEM vendors do not necessarily need to wait for patches to propagate to Qualcomm stable. For example,

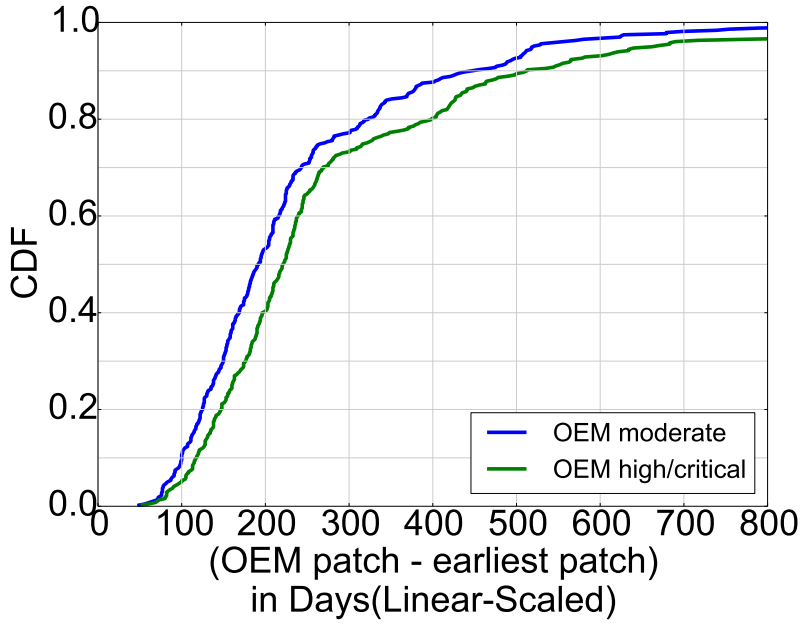

Figure 2.12: End-to-end delay between earliest patch and OEM phones

they could be notified by Google earlier.

Next, we also plot the end-to-end delay in Figure 2.11 by adding up delays in each propagation layer in the whole ecosystem. Here the earliest patch is either Linux mainline or Qualcomm, depending on whether the vulnerability is originated from Linux or Qualcomm. Generally, both cases incur significant delays with Linux vulnerabilities being generally worse. This is understandable because a Linux patch naturally has a longer propagation chain compared to a Qualcomm patch. As we can see, more than half of the Linux CVEs are delayed for 200 days or more, and 10% to 30% of CVEs are delayed for more than a year. This is an unacceptably long delay that allows experienced hackers to craft exploits

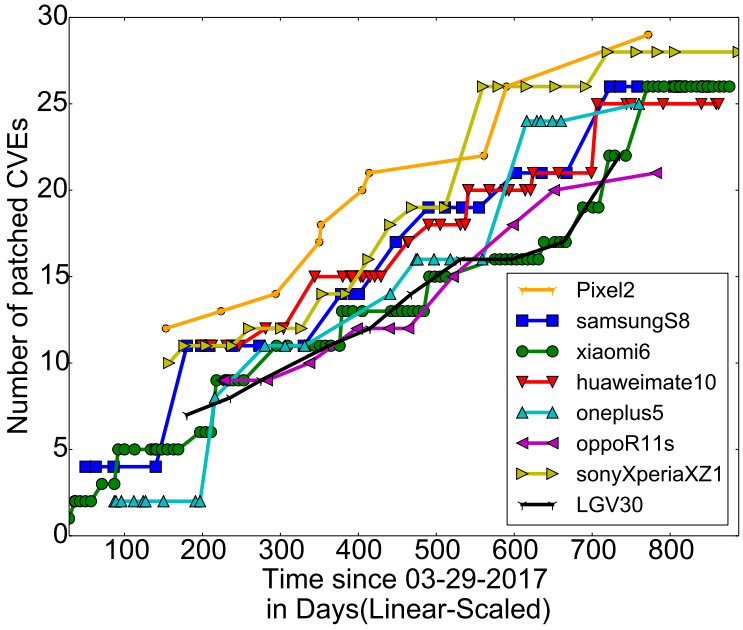

Figure 2.13: Different OEM vendor comparison

against unpatched OEM devices. CVE-2019-2215 is one such example [101].

Next, we analyze a number of factors that might influence the patch delays in OEM phones.

• **Vulnerability severity.** Intuitively, more severe vulnerabilities should be patched sooner rather than later by OEM vendors (or upstream). However, as shown in Figure 2.12, the result is not supportive. Specifically, we plot the distribution of end-to-end patch propagation delays by vulnerability severity levels. In §2.6, we will offer a much more detailed explanation of the phenomenon (after reaching out to Google). Note that there are only 33 critical CVEs from the security bulletin, and 30 of them are very old (originally patched before 2017) not applicable to many of the new OEM devices. Thus we combine them with high severity CVEs.

• **Name brand.** To do a fair comparison, we sample 8 phones from 8 first-tier companies

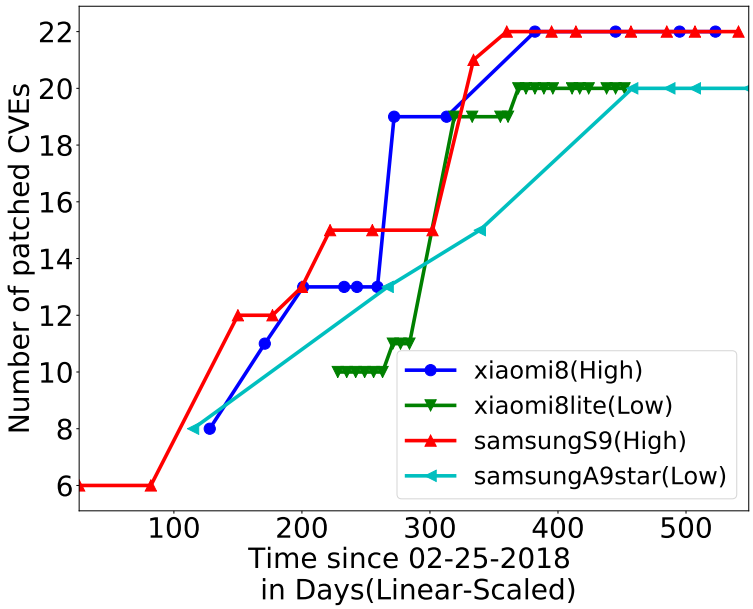

Figure 2.14: High/low-end phone comparison

which are all high-end and released in 2017: Google Pixel2, Samsung S8, Xiaomi Mi 6, Huawei Mate 10, Oneplus 5, Oppo R11s, SONY Xperia XZ1 and LG V30. Their corresponding kernel versions are also the same — 4.4.y. We only compare the CVEs that affected all target phones and ignore the CVEs patched beforehand. As seen in Figure 2.13, the results show that Google Pixel 2 and SONY clearly did the best. In contrast, Xiaomi, Oppo, and LG are the slowest.

• **High-end vs. Low-end.** This may be an expected result as companies tend to devote more resources to their flagship phones. Figure 2.14 shows the comparison between high-end phones (Mi 8, Galaxy S9) and low-end phones (Mi8 Lite, Galaxy A9 star) in Samsung and Xiaomi.

• **Geographic locations and carriers.** We did a small sample analysis of Samsung and Huawei phones, and the results show that the same kind of phone (only with minor

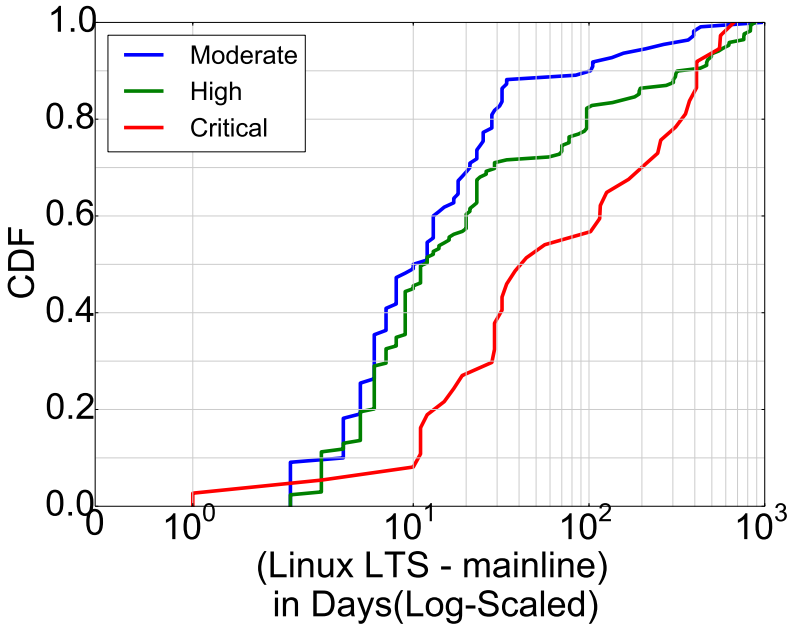

Figure 2.15: Patch delays from Linux mainline to LTS (by severity)

adjustments, *e.g.,* for local carriers) in different regions got patched at the same time in most cases, with about only 10 percent of the cases being slightly different.

• **Time after release.** Android devices are known to have a relatively short support lifetime, *e.g.,* Google phones now offer mostly 3 years of security updates [99]. In practice, most phones (especially high-end ones) do indeed enjoy at least 2 years of support. A major exception is Xiaomi's Redmi 4, a popular low-end phone popular in China and India. It was released in 2017 and still had some updates (*i.e.,* new firmware images) until March 2019. However, surprisingly it stopped patching any security vulnerabilities since early 2018 (less than a year).

2.6 Causes of Patch Delays

So far, we have quantified the patch delays in the Android kernel ecosystem mostly in a "blackbox" manner. However, other than blaming the long chain of patch propagation, we have not explored the reasons why the delays are so profound. They can be illuminating for future improvements in patching practices.

To achieve this goal, we collect additional information to help explain the rationale behind the patching practices by each participating party in the ecosystem. Specifically, we will analyze the security bulletins released by more organizations (Qualcomm), extract more details related to each patch commit, and reach out for information to the various parties including Google, Qualcomm, and Samsung.

From an intent point of view, a security patch can be applied in either of the two ways: *knowingly* or *unknowingly*. For example, an OEM vendor may be notified by Google about a serious security vulnerability and knowingly look for patches from upstream. On the other hand, Google may be blindly applying all upstream commits from Linux LTS to Android common branches, not knowing which are important security patches. Understanding the intent will provide valuable insight into the patching delays.

Based on this basic framework, we propose the following hypotheses to explain the slow patching.

(1) Even though the Android kernel ecosystem is largely open-source, the "knowledge of a security vulnerability" is often lacking and does not traverse the ecosystem fast enough, preventing security patches from being recognized and "knowingly" picked up by those who are affected (e.g., OEM vendors).

(2) A downstream kernel branch may have drifted from the upstream (*e.g.,* customization in downstream), it is not always possible to blindly apply all upstream commits (conflicts can arise). This may cause some kernels to lower the frequency to "sync" with upstream kernel branches, reducing the possibility of "unknowingly" patching a vulnerability in time.

To validate the hypotheses, we look into detailed commit log of kernel repositories. As all kernel repos (*i.e.,* Linux, Qualcomm, and Android common) are managed by git, we are able to differentiate through the commit log whether an upstream patch is knowingly "cherry-picked" or unknowingly "merged" (together with a stream of commits) into a downstream kernel branch. They correspond to the command `git cherry-pick <upstream-commit>` and `git merge <upstream-commit>` respectively. The semantic of cherry-pick is to pick a specific upstream commit and port it over to downstream, whereas merge pulls all the commits since last divergence up to `<upstream-commit>`.

Cherry-pick is more flexible as it can patch specific vulnerabilities without influencing other features. However, it requires knowledge about which upstream commit corresponds to an important security patch. In other words, the downstream must either be notified about the patch or identify the security issue proactively.

Merge treats all upstream commits equally and does not differentiate between security patches (severe or not) and other bug fixes. If done frequently enough, patch delays can be effectively reduced. The drawback is that manual resolution is needed when merge conflicts occur.

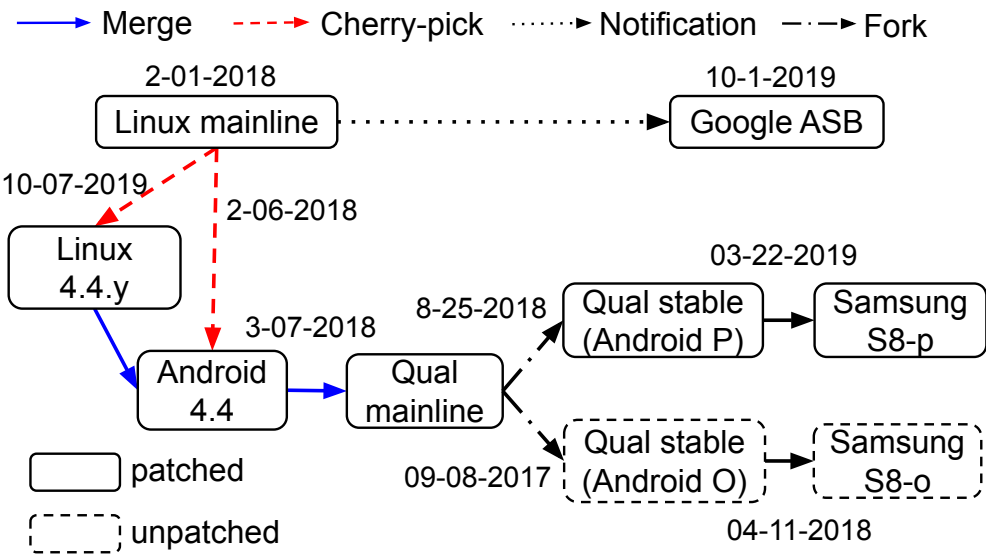

Figure 2.16: Case study: CVE-2019-2215

Similar to merge, fork is sometimes used by a downstream to become a clone of an upstream. This way, the downstream automatically inherits all the patches applied in the upstream at the time of fork. The drawback is if any customization is made in downstream, however, it needs to be ported over to the newly forked branch.

Next, we use a case study of a known CVE to demonstrate when these patch operations are performed, and how they can help explain the patch delays.

Case study. In Figure 2.16, we illustrate the above patch operations using CVE-2019-2215, a serious vulnerability that allows rooting [102] which was originally patched in Linux mainline on 2/1/2018. The cherry-pick by Linux 4.4 LTS occurred on 10/7/2019 with a long delay. Notably, Google's Android common 4.4 branch proactively cherry-picked the patch from Linux mainline on 2/6/2018 (bypassing its direct upstream). Unfortunately, Google does not appear to be aware of how serious the vulnerability is, evident by the

extremely late Android security bulletin announcement on 10/5/2019 (an 18 months delay) and Google's public statement admitting them being informed by the project zero team on 9/26/2019 [101]. It is also worth noting that no CVE was issued prior to the point. During this time, Qualcomm was uninformed about the vulnerability either. Its stable branch `kernel.lnx.4.4.r27-rel` did not cherry-pick the patch, leaving the corresponding Samsung S8-Oreo (Android 8.x) to be vulnerable all this time [102].

On the other hand, Qualcomm stable branch `kernel.lnx.4.4.r35-rel`, representing the same chipset with an upgraded Android Pie (9.x) had been merging updates from android-4.4 periodically (merge is preferred in Qualcomm stable prior to its release), thus patching the vulnerability on 3/7/2018. Luckily, when Samsung S8 upgraded its OS from Oreo to Pie, it forked from this stable branch, inheriting the patch unknowingly. Unfortunately, other OEM phones using the same chipset (and staying on Android Oreo) will remain vulnerable unless they cherry-pick patches elsewhere. In fact, we have checked that `kernel.lnx.4.4.r27-rel` never bothered to apply the patch until the end of its lifetime on 1/22/2020.

The case study gives us good insight on how the patching process is like in the ecosystem. Next, we will generalize the insight by analyzing each step of the propagation closely and offer takeaways and suggestions on how to improve the ecosystem.

1. Linux community. Linux vulnerabilities are always first patched in Linux mainline and then cherry-picked by downstream branches. Since Linux stable/LTS branches aim to operate as reliably and stably as possible, there is a formal set of rules guiding the

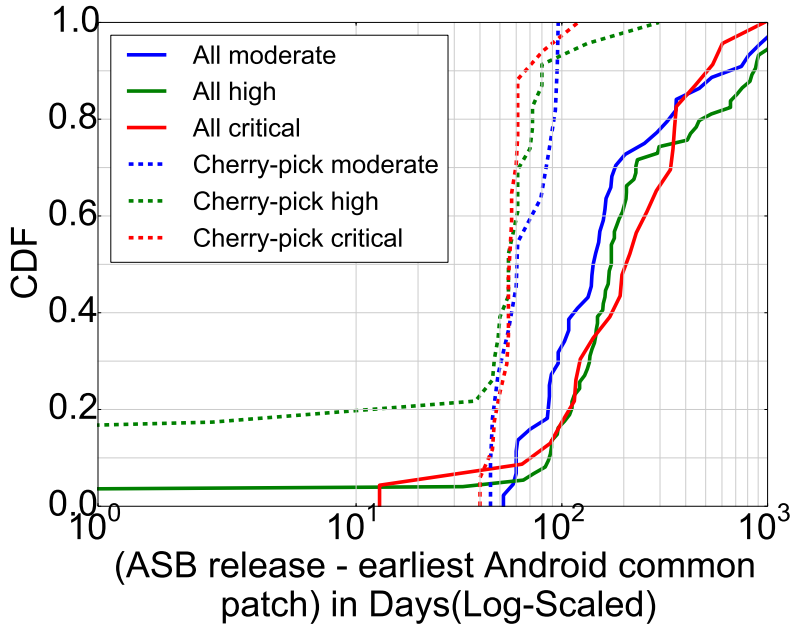

Figure 2.17: Notification delays of Linux CVEs (by severity)

cherry-pick of upstream patches [18], *e.g.,* "it cannot be bigger than 100 lines, with context; it must fix a real bug that bothers people, ... a real security issue".

Thanks to the close collaboration between Linux mainline and stable maintainers and the fact they belong to the same community, patch delays between the two are generally small. The outlier 3.18.y was noticeably slower than others. It turns out that other than the fact that it is an older branch, it was never meant to be an LTS branch. However, due to popular demand from Android kernels which decide to fork from 3.18.y, it remains actively maintained for much longer than originally intended. This may partially explain the slow cherry-pick of upstream patches. In other LTS branches, patch delays are generally small despite a long tail.

Unfortunately, due to the general principle followed by Linux that "a bug is a

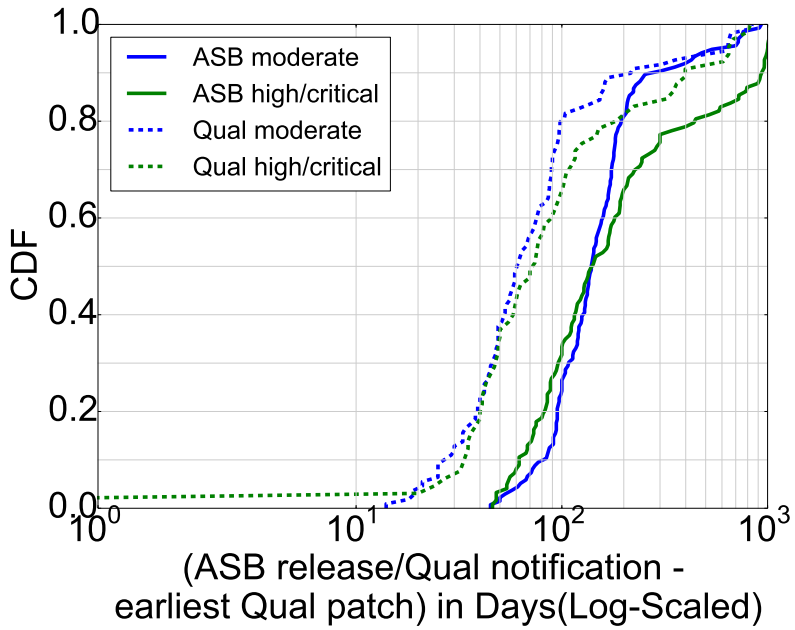

Figure 2.18: Notification delays of Qualcomm CVEs (by severity

bug" [33], oftentimes the Linux community does not realize whether a bug is truly an exploitable security bug until much later. By convention, security patches in Linux are not labeled as such in the public commit logs [100]. This creates a situation where Linux LTS maintainers are not even aware of the impact of those vulnerabilities. As supporting evidence shown in Figure 2.15, counterintuitively, CVEs that are (later) rated as critical and high by Google turn out to take noticeably longer time for Linux to patch, indicating the lack of knowledge by Linux. In fact, we find 17 out of 37 patches for critical vulnerabilities were initially missed in the initial "train" of cherry-picked patches, as they appear "out-of-order" with respect to other cherry-picked patches.

Even when Linux is aware of a security vulnerability, *e.g.,* notified by an external party via the private vulnerability reporting mailing list, `security@kernel.org`, this

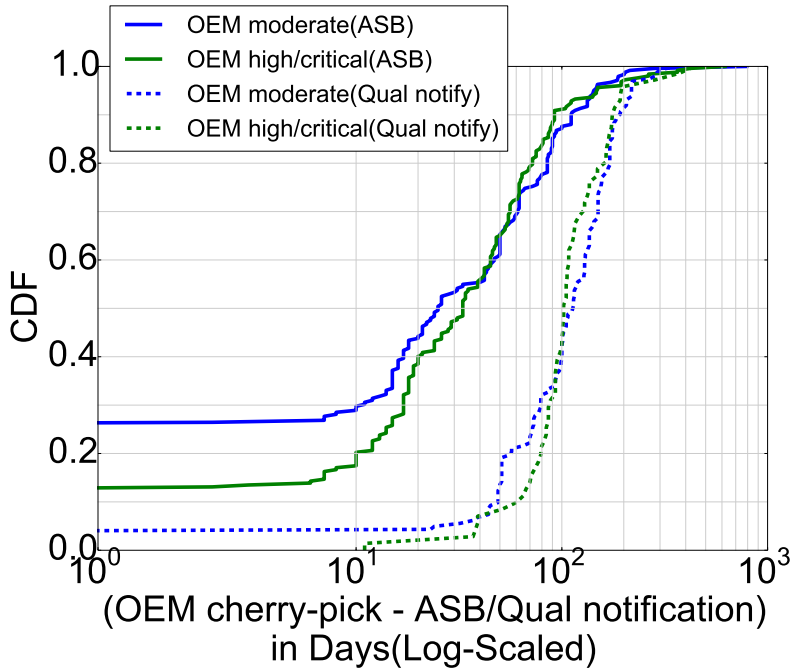

Figure 2.19: Post-notification delays of cherry-picked patches (by severity)

knowledge may or may not propagate internally to Linux LTS maintainers. In addition, as Linux's commits are often intentionally opaque [100], the knowledge is almost definitely lost outside of Linux, preventing downstream kernels from cherry-picking the corresponding patches timely. The only publicly available mechanism to document such knowledge is the CVE database. However, it is known to be incomplete and takes a long time to assign a CVE number and to update the entry [33].

Therefore, a better mechanism to track security issues is needed. Specifically, for the vulnerabilities that are reported to Linux through its private mailing list, we argue that it is a big missed opportunity where Linux has already triaged the bug and can clearly label the corresponding fixes as security-critical to help the downstream kernel (this is much more efficient than the CVE mechanism). For other bug fixes, we call for better tools to

Propagation step	3.18	4.4	4.9	4.14
LTS ->Android	63/106	74/105	70/74	30/31
Android ->Qualcomm	26/95	93/109	72/74	61/66

Table 2.4: The ratio of CVEs patched by merge

automatically reason about the nature of a bug and determine if it has serious security implications — a recent tool has been developed by Wu et al. [195].

2. Google. Android common kernels are forked from Linux stable/LTS initially and then add Android-specific changes on top (sometimes referred to as "out-of-tree" code). Over the years, Google has been upstreaming much of its code to Linux mainline and reducing such "out-of-tree" code [114]. This allows Android common kernels to merge patches from Linux LTS with a delay of 0 day, a week, to a month sometimes, and only occasionally cherry-pick from Linux mainline directly for important security patches. This is evident in Table 2.4 which shows the exact numbers of patches merged vs. cherry-picked. Note that 3.18 and 4.4 are exceptions as most of the patches in the beginning were cherry-picked from Linux mainline where the delays are less predictable (some are creating negative delays compared to Linux LTS).

In addition to keeping its own Android common kernels up-to-date, Google has another important responsibility to notify OEM vendors about security patches. While the exact notification date is mostly not made public, according to Google, it typically goes out at least a month prior to the information appearing on the security bulletin [8]. Surprisingly, as Figure 2.17 shows, in the majority of the CVEs, it takes anywhere from 100 to 500 days for the details to appear on the security bulletin (note that the actual notification should be at least 30 days earlier). In the extreme 20% of the CVEs, it takes 500 days or more. We

believe this is due to the fact that Google is not really aware of which of the merged patches are security-critical — indeed the delays shown in the figure do not appear correlated with the severity of vulnerabilities.

In the same figure, we also show the notification delays of CVEs where Google knowingly cherry-picked important security patches. Indeed, the delays are noticeably smaller. This indicates the lack of knowledge is the culprit again, supporting our hypotheses. There is still not too much difference based on vulnerability severity levels. After finishing the analysis, we also confirmed with Google that this is expected as their pipeline does not distinguish severity levels by design. Every month, all issues rated above the threshold and known to Google, *e.g.,* moderate and above, are worked on together in a batch. Exceptions occur only under extraordinary circumstances where disclosure of a serious vulnerability is imminent.

In general, for vulnerabilities that originate in Linux, better and more automated vulnerability triage seems to be a key capability that can benefit Google. Manually sifting through merged upstream commits and narrowing down to the handful that eventually appears on the Android security bulletin can be prohibitively expensive. Alternatively, if Linux has done the triage already, Google can benefit directly from the knowledge, *e.g.,* through tighter collaboration.

For vulnerabilities that originate in Qualcomm, Google should have the first-hand knowledge already — they are almost always informed by either Qualcomm or external parties about the specifics. In such cases, the notification to OEM vendors should be as swiftly as possible, which unfortunately is not the case as we will discuss later in the section.

3. Qualcomm. Qualcomm maintains many more branches compared to Linux and Google and the overhead of patch tracking and management goes up. However, we find its mainline branches are maintained in a similar fashion to Android common. As seen in Table 2.4, mainlines primarily merge commits from Android common and only occasionally cherry-picks patches from Linux directly. One difference is the merge frequency is generally lower than that of Android common, resulting in longer delays as shown in Figure 2.7.

On the other hand, Qualcomm stable branches are maintained differently. After they are forked from a mainline and labeled as "release", only cherry-picks are performed. This creates the same paradox that even though Qualcomm mainlines merge patches relatively timely, the developers are not aware of the security-critical nature of these patches. As a result, it can take Qualcomm stables a long time to cherry-pick the patches. Indeed, Figure 2.8 illustrates the dramatic delay. Shockingly enough, after we reach out to Qualcomm about the delays, their response indicates that this is because stable branches often receive Linux-specific patches only when customers ask for them explicitly.

In principle, even if Qualcomm is interested in proactively patching Linux vulnerabilities, the knowledge gap needs to be bridged by Linux (*e.g.,* labeling the security nature of a patch). However, Qualcomm can do its part by merging more patches to stable branches without distinguishing their nature, despite the fact that Qualcomm stables are designed to include bug fixes only. This is because Qualcomm stables are already based on Android common branches and indirectly from Linux stable/LTS, which commit necessary bug fixes only (no new features). Interestingly, we observe two recent stable branches based on Android 10, namely `kernel.lnx.4.9.r34-rel` and

`kernel.lnx.4.9.r30-rel` in Qualcomm follow this very strategy.

In contrast, for vulnerabilities that originate in Qualcomm kernels, we know that they are patched much more timely in stable branches (see Figure 2.8). In such cases, Qualcomm is likely already aware of the nature of the bugs — most are described as externally reported or internally discovered during auditing. Thus Qualcomm should be able to notify OEM vendors as soon as patches are available. Unfortunately, after collecting data from Qualcomm's security bulletin (released monthly since Sep 2017), we found that the delay between the earliest patch and its own notification date is not ideal (median delay: 63 days, mean delay: 130 days), as shown in Figure 2.18 (surprisingly indiscriminative of the vulnerability severity again). Note that we combine high/critical CVEs into one line here because there are only three critical Qualcomm kernel CVEs since the inception of Qualcomm's security bulletin.

After confirming with Qualcomm, we know that the customer notification is sent out (to all OEM vendors) only after fixes have been widely propagated on affected branches. However, we believe the notification process can be more agile — a subset of OEM vendors can be notified as soon as their corresponding branches have the patches ready. Even better, oftentimes the patches are not really different across branches, Qualcomm can simply notify all customers as soon as the earliest patch is ready and OEM vendors can make an early decision (*e.g.,* testing the patch independently before applying). This way, the major bottleneck of late notification can be mitigated.

According to the same figure, there is another delay of two to three months before Google publishes these CVEs on its security bulletin. Since most OEM vendors follow

Google's monthly schedule to update security patch level, OEM patches will be unnecessarily delayed.

4. OEM phones. To understand how patching is performed on OEM kernels, we refer to the Pixel source branches as well as an Oneplus repo that happened to contain the complete commit history. We observe that these kernels cherry-pick patches from Qualcomm (either mainline or stable) and even Linux sometimes. In addition, when OEM vendors decide to upgrade the Android OS (*e.g.,* Android Oreo to Android Pie), they usually abandon the old branch and develop another stable branch (forking from upstream) that corresponds to the new Android OS (as the case study about Samsung S8 showed). We can infer that other OEM vendors follow the same strategy of (1) cherry-picking instead of merging, and (2) forking when upgrading. This is because (1) the firmware images often skip upstream patches (so it is unlikely performing `git merge`), and (2) OS upgrades always happen together with the kernel version updates, which is also the case with Qualcomm stable branches — OS upgrades lead to a new stable branch with an advanced kernel version. In addition, we always observe a large number of kernel patches applied when the firmware is upgraded to a new Android OS.

Specifically, depending on the exact phone model, 30% to 75% of CVEs can be patched through forking a new branch from upstream. This is not a healthy number because Android OS upgrades usually happen on a yearly basis and not to mention that there are often additional delays for these upgrades to reach user devices (*e.g.,* carrier delays). Clearly, more patches should have been cherry-picked in between upgrades.

For the cherry-picked patches, we consider them timely if they are applied within

a reasonable amount of time after Google or Qualcomm notify the OEMs, which is typically expected to be a month or two. Unfortunately, OEM vendors are often significantly behind the schedule. As Figure 2.19 shows, 80% of the Qualcomm CVEs take OEMs 100 days or more to deploy corresponding patches. This is likely because OEM vendors ignore Qualcomm's notifications and prefer to follow the monthly updated security patch level set by Google. We contacted Samsung and confirmed that OEMs are bound to follow Android's monthly bulletin while no such strict requirements exist for Qualcomm. This is reflected in the figure where more than 50% of the CVEs take OEMs less than a month (sometimes even beforehand) to patch after the Android security bulletin publication (which is within the expectations [47]). As we can see, Google's notification plays a huge role in getting OEMs to patch.

We note that there is a small fraction of patches (roughly 5%) delayed for 200 days or more after Google's security bulletin is published. This is not only due to slow and infrequent security updates by some devices but also occasionally skipped CVEs (out of the ones published together in a month). For example, we find that Samsung S8 has skipped nothing but CVE-2018-13900 from Google's Feb 2019's security bulletin, which interestingly got patched eventually in 2020. Finally, from Figure 2.19, we do not find significant correlation between the severity of vulnerabilities and timeliness of patches being cherry-picked by OEMs. Note that the number of critical cherry-picked patches by OEMs is very limited, especially for some new phones, thus we combine high and critical ones into a single line. In fact, CVE-2018-13900 is a high severity vulnerability yet skipped by Samsung S8.

To improve the situation, OEM vendors should obviously react more timely to the earliest notification, *e.g.,* Qualcomm. Furthermore, similar to what we suggest for Qualcomm, OEM vendors can consider merging patches directly from upstream instead of cherry-picking them. We also hope that high-end and low-end phones can be treated equally, as we show low-end phones tend to receive patches more slowly in Figure 2.14. At the end of the day, we believe a better and more automated patching/testing process will help.

Summary. Overall, the analysis supports our hypothesis and we propose three general areas that need improvement.

More efficient triage systems. The triage process of security vulnerabilities today is largely manual. This is evident in the case study where the initial bug fix made in Linux mainline was never treated seriously enough by the rest of the ecosystem (Linux LTS failed to cherry-pick it also). Better automated reasoning tools (*e.g.,* [195]) can assist the developers in identifying security-critical bugs and take actions accordingly.

More efficient knowledge propagation. Unfortunately, even when the knowledge of an important security vulnerability does become available in one party, it either does not have a good mechanism to propagate the information (*e.g.,* Linux), or propagate the information in a delayed manner (*e.g.,* notification by Google and Qualcomm). In addition, sometimes it is beneficial to propagate the knowledge in the reverse direction (*e.g.,* some patches shown to be applied in Google before Linux LTS). Ideally, this process should be more automated to reduce delay.

Cleanly separate the changes made in downstreams. Current patching practices

in downstreams largely rely on cherry-picking, *i.e.,* Linux LTS, Qualcomm stables, and OEMs. If a downstream kernel can cleanly separate its customization code from the upstream, or even better, upstream its customization (as is the case with Google[114]), the responsibility of patching upstream vulnerabilities can be completely automated with merging, *i.e.,* Android common and Qualcomm mainlines. A downstream kernel can simply merge automatically and fix security issues unknowingly.

2.7 Discussion

Unpatched kernels. By design, patch presence test is unable to equate the absence of patches with the target "being vulnerable". Throughout our measurements, we observe many cases where the downstream kernels never apply patches from upstream. However, this could simply mean that the downstream kernel is not affected by the upstream vulnerability, *e.g.,* due to customization of the vulnerable function. This is why we focus on the patched cases only, because it implies the downstream kernels are affected.

Further delays after the OEM patches. Our patch propagation measurement stops at the kernel compilation (build) dates. However, in practice, there are additional delays before the OEM updates can arrive at a user device. They include carrier certification delays (for carrier-locked phones), and users intentionally delaying the firmware update even if it is already available through OTA. Unfortunately, such delays are hard to quantify and we consider them out of scope. To get a basic sense of carrier certification delays, we manage to find the LG V30/Samsung S7/Samsung S8 on T-Mobile websites and SamsungS7/SamsungS8 on ATT websites that appear to publish the firmware release

date. The average delay between built and release is about 20 days. To draw any meaningful conclusions though, a large-scale analysis needs to be done across more devices and carriers.

Chipset vendors other than Qualcomm. In addition to Qualcomm, other major SoC vendors include MediaTek, Kirin, and Exynos. Unfortunately, none of these vendors provides the complete git repositories for their recent chipsets. In addition, the CVEs specific to Kirin and Exynos chipsets are published only on Huawei's and Samsung's official websites but no links exist to the corresponding patches. Together, they represent a hurdle for any external party to track their patches. We suspect reverse engineering on the firmware images will be the only way to analyze the presence and absence of patches.

2.8 Related Work

Code similarity at the source and binary level. To conduct our measurement we need the ability to accurately test the patch presence at both source level (*e.g.,* the source code of the phone kernel is released) and binary level (*e.g.,* only ROMs are available for the target phone). There exist a large body of work aiming to compute the source/binary code similarity (*e.g.,* to find similar functions as a given vulnerable one), using a variety of source and binary level features [57, 110, 109, 152].

In theory, these work can be used to test the patch presence by computing a target function's similarity to the patched/unpatched functions). Unfortunately, similarity-based approaches are fundamentally fuzzy and not suitable to capture the essence of a security patch which often makes only very small changes to patched functions and can still look similar to the unpatched version of the function. Tuning the similarity-based approach for

patch presence test is an interesting but orthogonal problem.

Binary patch presence test. FIBER [207] is a state-of-art open-source tool to test the patch presence in binaries with the aid of the fine-grained source level patch information. It generates binary signatures that accurately capture the syntax and semantic information of the patch change sites, and then matches them in the target binary. It suits our needs perfectly and therefore we leverage and build on top of FIBER to test the patch presence for over 600 Android ROMs. To ensure that it works well in our large-scale measurement, we enhance the original FIBER to overcome several of its technical weaknesses as detailed in §2.4.3.

Android security patch investigation. Farhang *et al.* [88] have recently conducted a measurement on Android security patches, including both user and kernel components, with some minor overlap with this paper. In particular, they also analyzed the delay from the patch date (linked from the security bulletin which we now know is often not the earliest date) to the release date on the bulletin and observed a large delay. However, this represents only a small part of the picture of the end-to-end patch propagation in the ecosystem all the way from the upstream Linux to the end Android devices. Specifically, they do not attempt to locate patches in the source or binary at all. Thus they cannot find the bottleneck of patch delay. On the other hand, we not only showed where the bottleneck is but also explained why they exist with actionable insights and takeaways. More importantly, we also give suggestions on how to improve the patch propagation in the ecosystem.

Patch and vulnerability lifecycle analysis. There exist a number of measurement studies focusing on various aspects of patch propagation in open-source software. Li *et*

al. [128], Shahzad *et al.* [165] and Frei *et al.* [91] performed large-scale measurements regarding the vulnerability lifecycle and the patching timeliness, based on publicly available information collected from data sources like CVE databases [19] and open-source repositories. Some of them focus on specific open-source projects, like Farhang *et al.* [88] focusing on Android and Ozment *et al.* [148] targeting FreeBSD. No analysis has been dedicated to the Android kernel ecosystem which involves the analysis of multiple parties in depth and the analysis of source and binary kernels.

2.9 Conclusion

In this paper, we delved deep into the Android kernel patch ecosystem, revealing the relationship among different parties as well as the bottleneck in patch propagation. This represents a first data point to measure such a huge, decentralized, fragmented, and yet collaborative project. We also analyze that the study is worthwhile in identifying deficiencies and opportunities to better manage such a project in the future.

Chapter 3

SymBisect: Accurate Bisection for Fuzzer-Exposed Vulnerabilities

Abstract

The popularity of fuzzing has led to its tight integration into the software development process as a routine part of the build and test, i.e., continuous fuzzing. This has resulted in a substantial increase in the reporting of bugs in open-source software, including the Linux kernel. To keep up with the volume of bugs, it is crucial to automatically analyze the bugs to assist developers and maintainers. Bug bisection, i.e., locating the commit that introduced a vulnerability, is one such analysis that can reveal the range of affected software versions and help bug prioritization and patching. However, existing automated solutions fall short in a number of ways: most of them either (1) directly run the same PoC on older software versions without adapting to changes in bug-triggering conditions and are prone to broken dynamic environments or (2) require

patches that may not be available when the bug is discovered. In this work, we take a different approach to looking for evidence of fuzzer-exposed vulnerabilities by looking for the underlying bug logic. In this way, we can perform bug bisection much more precisely and accurately. Specifically, we apply under-constrained symbolic execution with several principled guiding techniques to search for the presence of the bug logic efficiently. We show that our approach achieves significantly better accuracy than the state-of-the-art solution by 16% (from 74.7% to 90.7%).

3.1 Introduction

In recent years, large-scale programs such as the Linux kernel are being continuously fuzzed for the purpose of improving code quality and security [98, 55, 182, 86, 150, 103]. Such continuous fuzzing systems have been shown highly effective in identifying new bugs, e.g., syzbot [97] reports thousands of bugs in the Linux kernel.

While fuzzing is highly effective, this poses large workload to software developers and maintainers, as the continuous stream of bugs requires various analysis, e.g., bug triage and patching, that is often done largely manually today [132]. This is a hard problem as we already see over 8,000 bugs found by syzbot are auto-closed due to the lack of human investigations [97]. Thus, automating the analysis of fuzzer-exposed bugs is a worthwhile goal.

One important analysis that needs automation is bug bisection, i.e., the process of identifying commit that introduced a bug (also called vulnerability-contributing commits, or bug-inducing commits). It proves instrumental in various aspects. For example, it can

help developers and maintainers understand the bug and facilitate patch development [46]; it can also pinpoint the vulnerable software versions to inform users about whether they need to worry about updating their software [58, 26].

To achieve this goal, researchers have proposed several automated approaches, but unfortunately they all have significant shortcomings.

The first type of approach directly executes the original PoC on older software versions to see which version would still crash after running the PoC. However, it is reported that such a dynamic solution suffers from several issues [25]: 1) Broken dynamic environment (e.g., build or runtime errors) leading to versions being skipped. 2) Accidental triggering of unrelated bugs. 3) Changes in the underlying bug-triggering condition.

The second type of approach requires patches, which may not be available at the time of the bug discovery. Even if the patches are available, such static solutions rely on heuristics which are inherently imprecise [50, 159]. For example, when given the code diff in a patch, their solutions consider the bug-inducing commit to be the one that introduces one or more lines in the code diff [58, 194]. However, such a solution does not take into account the bug-triggering conditions and can miss important details that are outside of the scope of the code diff in the patch.

Motivated by the above deficiencies, we take a different approach from the traditional methods. In particular, we aim to reason about the *presence of vulnerability logic* through static code analysis. Fundamentally, our approach investigates many more possible inputs beyond what's included in the original PoC. Furthermore, static methods

can effectively circumvent a series of problems caused by the broken dynamic environments such as build errors. Finally, it does not require the development of patches in advance. To this end, we leverage **symbolic reasoning** which is the most precise way of confirming the presence of a vulnerability statically. A crucial characteristic of this approach is that it can automatically distinguish significant changes from vulnerability-irrelevant changes and effectively eliminate the influence of vulnerability-irrelevant changes on the results.

More specifically, we apply under-constrained symbolic execution [157] in different software versions to precisely identify the presence/absence of the same *vulnerablility logic* that is inherited from the released PoC. Then with a simple binary search algorithm, we can pinpoint the commit that introduced the vulnerability. To address the scalability challenges of symbolic execution, we leverage the trace associated with the PoC to guide the symbolic execution.

Following the methodology proposed in this paper, we apply it to the context of Linux kernel and the corresponding continuous fuzzing platform syzbot [97]. We show that it significantly outperforms state-of-the-art approaches in terms of accurately determining the vulnerable versions of bugs found with fuzzing. We summarize our contributions as follows:

- We developed a novel and drastically different solution of an automatic bisection tool called SymBisect, targeting fuzzer-exposed vulnerabilities. Our method is precise as it relies on looking for the presence of key vulnerability logic represented by symbolic formulas. We have implemented SymBisect for Linux kernel bugs reported on syzbot.

We open-sourced the solution to facilitate the reproduction of results and further research [24].

- We proposed a new method to address the scalability problem in under-constraint symbolic execution in the Linux kernel. Our insight is that in the specific context of fuzzing results, we are able to use the knowledge of the vulnerability from the PoC to guide the symbolic execution in a principled fashion.
- We evaluated the performance of SymBisect against other state-of-the-art methods. We demonstrate that it not only achieves much higher accuracy than the PoC-based bisection but even outperforms the methods dependent on the presence of patches. Specifically, it can identify 83% of the vulnerable versions that elude detection in the PoC bisection implemented by Syzbot.

3.2 Background and Motivation

In recent years, fuzzing has played a significant role in discovering vulnerabilities in the Linux kernel [97, 98]. However, manual analysis of the extensive results generated by fuzzing has placed a tremendous burden on maintainers [17]. Automatic analysis of fuzzing results, such as identifying vulnerable versions and simultaneously identifying when vulnerabilities were introduced, is highly beneficial for understanding the logic of vulnerabilities, developing patches, notifying the respective maintainers, and backporting patches to vulnerable Long-Term-Support branches. For example, Rui Abreu *et al.* observed that automating bug bisection that pinpoints the bug-inducing commits can speed up fixing fuzzer-exposed bugs in Google's proprietary code on average by a factor of

2.23 [46]. In this paper, we define

PoC-based bisection. The most straightforward approach is to dynamically re-execute the PoC that triggered the bug on older commits. This method is employed by the continuous fuzzing platform syzbot [97]. Specifically, syzbot starts bisection by running the same PoC with the commit on which the bug was discovered, ensures that it can reproduce the bug, and then goes back release-by-release (e.g., v5.4 to v5.3) to pinpoint the earliest release without the kernel crash (again using the same PoC). The predicate for bisection is binary (crash vs. no crash), not trying to differentiate between different crashes. This is intentional because bugs can manifest in a different ways (under different bug titles) [25]. However, this inevitably introduces false positives as unrelated bugs can sometimes be triggered. In fact, a small-scale analysis showed that unrelated bugs being triggered contributed to 66% of incorrect bisection [26]. In addition, such an approach also leads to false negatives, i.e., failing to report a kernel version being affected by the bug during bisection. They can be due to build/boot errors, bugs that are difficult to reproduce, and failing to account for changes in bug-triggering conditions (no adaptation in the original PoC). Overall, a previous small-scale study conducted by the syzbot team concludes that the bisection accuracy is only about 50% [26], highlighting the need for a better solution.

Patch-based bisection. The basic idea of SZZ is to identify the bug-inducing commit by tracing the modified lines in the patch back to the most recent commit that introduced the lines. This method is static and effectively assumes the source lines removed or changed by the patch are responsible for introducing the bug. The SZZ algorithm has

many variations, among which VSZZ [58] is the latest improvement aimed specifically at vulnerabilities (instead of general bugs). VSZZ modifies SZZ slightly by tracing back the commit history to the earliest commit (as opposed to the most recent) that introduces the deleted lines of a patch. However, such methods require patches input, which are not available at the time of bug discovery. Furthermore, the SZZ algorithm and its variants are fundamentally heuristics and their accuracies are limited [50]. Finally, they are unable to handle patches with only added lines [58], which are quite common in security patches (e.g., adding a bounds check).

Our insight. This motivates us to develop a solution that looks for the presence of the vulnerability logic in target software versions.

Specifically, we propose to leverage under-constrained symbolic execution to effectively address the shortcomings of existing solutions. Compared to PoC-based bisection, our solution (1) is static, thus sidestepping the challenges stemming from broken dynamic environments; (2) focuses on the specific vulnerability, allowing it to overlook other unrelated bugs; (3) considers more possible inputs and execution paths, alleviating the concern of changes in the underlying bug-triggering conditions.

Compared to the existing patch-based methods, our solution (1) does not require patches, which are not available when a fuzzer first finds the bugs; (2) looks for the presence of vulnerability logic as opposed to syntatic information such as the presence of certain source lines or tokens; (3) inspects the vulnerability logic beyond the scope of patch functions, allowing a much more complete and informed validation compared to

heuristics that concentrate on only the code diff or the functions involved in patches.

3.3 Overview

In this section, we begin with a motivating example to provide a concise overview of why existing methods fall short and the intuition behind SymBisect. We will also discuss the main challenges of implementing our solution. Following that, we introduce the overall architecture of SymBisect.

3.3.1 Motivating Example

Figure 4.1 illustrates an integer overflow vulnerability that leads to an out-of-bounds memory access. Specifically, the **bug-inducing commit** modifies the function `htab_map_alloc()`, which in turn calls function `bpf_map_charge_init()` and function `prealloc_init()`. Prior to this commit, the function `bpf_map_charge_init()` had a check at line 6, which checked the variable `size` to prevent any potential integer overflow in `prealloc_init()`. However, the removal of this safeguard paved the way for the occurrence of an integer overflow. To mitigate this vulnerability, the subsequent **patch** introduced a type-casting operation at line 8 within `prealloc_init()`, effectively preventing the risk of integer overflow.

Prior PoC-based tool executed the released PoC in versions preceding the patch. However, in this case, it triggered an unrelated bug, leading the kernel to crash before it could access the function `htab_map_alloc()`. Consequently, this resulted in an imprecise bisection result — syzbot thinks the kernel version is vulnerable and keep checking even

The Bug-inducing Commit:

```
static struct bpf_map *htab_map_alloc(...)
1  - cost = S1*C1 + S2*S3;
     ......
2  - cost += S2*C2
3  - err = bpf_map_charge_init(..., cost);
4  - if (err)
   -     goto free_htab;
     ......
5    err = prealloc_init(...);

int bpf_map_charge_init(...,u64 size)
     ......
6    if (size >= U32_MAX - PAGE_SIZE)
        return -E2BIG;
```

The Patch:

```
static int prealloc_init(...)
     S3 = S3 + C2;
     ......
7  - htab->elems =bpf_map_area_alloc(S2*S3,
8  + htab->elems =bpf_map_area_alloc((u64)S2*S3,
```

```
S1: (u64)htab->n_buckets     C1: sizeof(struct bucket)
S2: (u64)htab->elem_size     C2: num_possible_cpus()
S3: htab->map.max_entries
```

Figure 3.1: The Bug-inducing commit and Patch of a vulnerability from syzbot

earlier versions.

Prior Patch-based tools derive various forms of signatures, primarily syntactic, from the patch function `prealloc_init()`. In this case, the bug-inducing commit does not alter the patch function. Consequently, these solutions are unable to capture the commit and fail to differentiate versions preceding and following the bug-inducing commit. This leads to incorrect identification of the bug-inducing commit.

Our solution symbolically executes the relevant functions until it reaches the target source line and evaluates the symbolic constraints to check whether an out-of-bound memory access can occur — we know it is an out-of-bound bug from the bug report. Specifically, the symbolic execution starts from the syscall entry that triggered the bug (available from the call stack in the bug report). By enlarging the analysis scope, our solution effectively explores more of the state space and is not confined to the patch function. It effectively addresses both the limitation of patch-based bisection and potential changes in the bug-triggering conditions. Additionally, by disregarding unrelated bugs, it resolves the issues associated with PoC-based bisection.

Figure 3.2 illustrates a portion of the symbolic execution process. In the non-vulnerable version (prior to the bug-inducing commit), the variable `cost` is assigned in lines 1 and 2, with a subsequent check at line 6. While there are two branches in line 6, only one of them leads to the vulnerability point. Within this path, symbolic execution identifies a crucial constraint: S2(S3+C2) + S1*C1 <U32_Max - 4096. This constraint ensures that the overflow condition S2(S3+C2) >U32_Max is never satisfied, preventing any subsequent out-of-bounds occurrences (the OOB section isn't depicted in the figure). Consequently, this version is deemed non-vulnerable, which is correct. In contrast, in the vulnerable versions, the critical check against the cost is removed. As a result, the overflow condition becomes solvable by the symbolic execution engine, leading to an out-of-bounds (OOB) situation later on. Accordingly, our solution correctly classified this version as vulnerable.

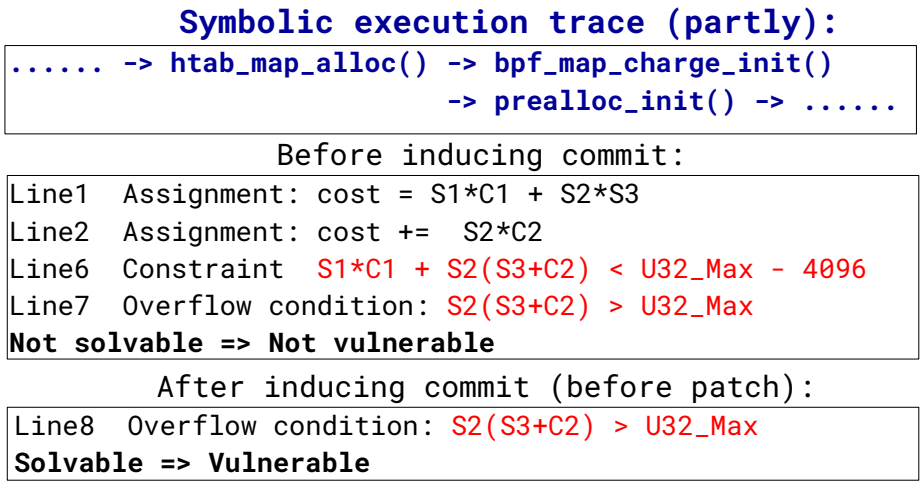

Figure 3.2: Vulnerability detection via symbolic execution

3.3.2 Challenges and Insights

Despite the advantages of using symbolic execution to confirm the presence of vulnerability logic precisely, symbolic execution also faces its own challenges.

Scalability concerns. In the motivating example, we showed only a segment of the symbolic execution in Figure 3.2. In reality, our solution will encounter many more functions (i.e., starting from the syscall entry) and accumulate many more symbolic constraints. This can lead to the classic scalability challenge for symbolic execution, as the number of forked states may grow exponentially as the execution progresses. This makes the solution seemingly ill-suited for a large scope of analysis, especially against large-scale software such as the Linux kernel. Previous methods deal with this problem by confining the scope of symbolic execution to one specific function [207] or utilizing concolic execution[67]. Nevertheless, these methods are unsuitable for our purpose: the existence of a vulnerability is not determined by a single function, and we do not want to over-constrain the possible inputs through concolic or concrete execution.

Key observation. We observe that, to overcome the above challenge, it is possible to leverage fine-grained trace-level information about how the vulnerability is manifested (e.g., where the vulnerability is triggered, and which functions are involved) in the reported version to guide the exploration in the target version. This information allows us to distinguish the key statements from the unrelated ones for a specific vulnerability. As an illustration, coverage data can help de-prioritize less relevant execution paths. By utilizing this approach, SymBisect effectively narrows the scope of exploration, thus enhancing efficiency and mitigating the scalability challenge of symbolic execution.

3.3.3 System Architecture

As illustrated in Figure 3.3, our tool, denoted as SymBisect, requires three essential inputs for its operation:

- The source code of the program on which the bug was reported – we refer to it as the reference version. This version should be compilable and bootable as the fuzzer has successfully found the bug on this version.
- Proof of Concept (PoC): This is the executable or script that can reliably trigger the vulnerability in the reference version of the program.
- The source code of the program in potentially vulnerable target versions: These are the other versions of the program that we want to assess for the same vulnerability.

SymBisect is designed for vulnerabilities found through fuzzing. So both the compilable and bootable source code of the reference program and the PoC are naturally available when a vulnerability is found via fuzzing. With such inputs, SymBisect bisects

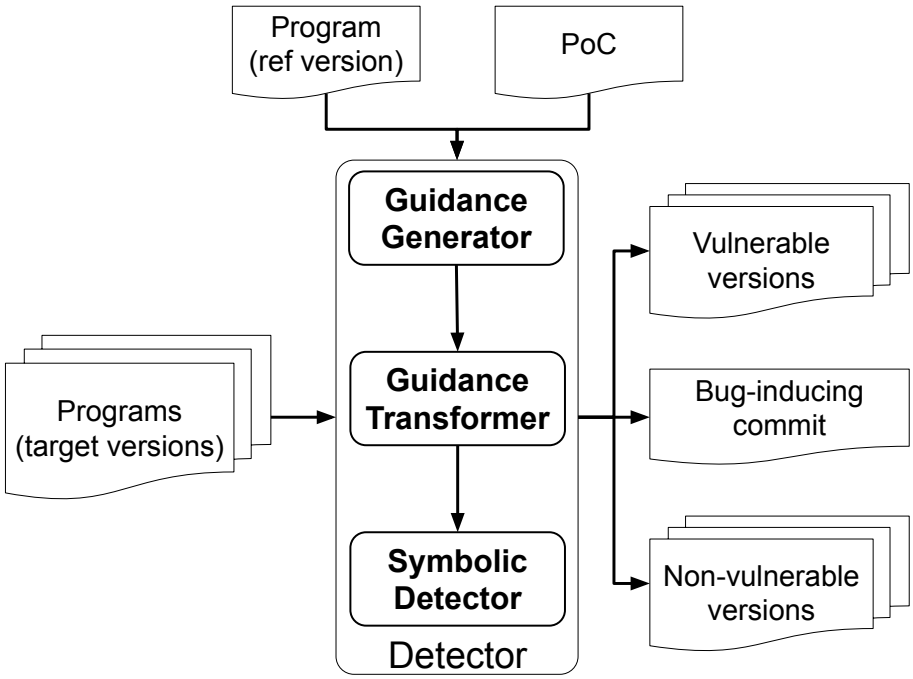

Figure 3.3: Overview of SymBisect

the bug in a fashion similar to syzbot (except that SymBisect is completely static). It first evaluates historical versions backwards – one major release version at a time (e.g., v5.5 and then v5.4). Through this iterative process, SymBisect can identify the boundary or range for which the bug-inducing commit falls under (e.g., between v5.4 and v5.5). Subsequently, SymBisect follow a simple binary search procedure to pinpoint the specific commit that introduced the bug.

SymBisect consists of three primary components, designed to accurately identify vulnerabilities while also addressing scalability issues:

Guidance Generator. SymBisect first runs a PoC to trigger the specific vulnerability in the reference version of the program, thereby collecting essential execution traces.

Utilizing these traces, SymBisect systematically produces three primary categories of guidance for subsequent symbolic detection. Firstly, SymBisect attempts to align the call stack trace (also called call trace in the syzbot bug report) of the execution on the target version to the one on the reference version (referred to as **Call Stack Guidance**) This effectively steers the exploration of symbolic execution towards the function where the vulnerability is observed. Secondly, . This is useful when there are a large number of possible execution paths that follow the same call stack. Lastly, SymBisect reuses the callees involved in indirect calls (referred as **Indirect Call Guidance**), thereby informing the symbolic detector to focus on a limited number of indirect call targets (as opposed to all possible ones computed using static analysis). More details are in §3.4.1.

Guidance Transformer. Upon identifying the above three kinds of guidance, SymBisect transforms them from the reference version into the target versions of the program. This enables a more efficient symbolic execution and a more accurate vulnerability detection process on these target versions. It's important to highlight that guidance translation between versions is done at the source code level, which remains stable and unaffected by compiler optimizations. To enhance the precision and robustness of those guidance when applied to differing target versions of the program, SymBisect employ multiple optimizations during the guidance transformer phase. More details about the guidance transformer will be provided in §3.4.2.

Symbolic Detector. The symbolic detector is a form of detector that can capture (or re-capture) the bugs that were reported by a fuzzer. The detector is applied to a target version, where it tracks all variables, especially symbolic variables, including the symbolic

sizes of allocated objects. However, instead of attempting to find all possible bugs during the exploration (which is clearly not scalable), we narrowly focus on the specific bug at hand, with the help of the aforementioned guidance. Throughout the execution, the symbolic detector leverages guidance from preceding phases. Specifically, it dynamically adjusts the execution state schedule, aiming to alleviate path explosion. Additionally, the detector refines the callees of indirect function calls based on prior indirect call guidance. For vulnerability detection, the symbolic detector relies on call trace guidance to ensure accurate detection of the same vulnerability previously identified. More details about the under-constrained symbolic detector can be found in §3.4.3.

3.4 SymBisect Design

In this section, we delve into the intricacies of SymBisect's design by dissecting each component, discussing the challenges encountered, and illustrating our corresponding solutions.

3.4.1 Guidance Generator

Overall, this components attempts to guide SymBisect when SymBisect executes the PoC in the reference version of the program to trigger a specific vulnerability and collects the execution trace. Then, SymBisect produces three categories of guidance from the execution traces, which we explain below.

Call Stack Guidance. The call stack guidance represents the state of the call stack at the moment a vulnerability is triggered. This information can be readily collected when

the corresponding bug is triggered in the reference version of the software under investigation. Utilizing the call stack guidance serves multiple purposes. First, it assists in identifying an appropriate **entry function** as the starting point for our symbolic detector. Second, it assists in pinpointing **the target line** where the vulnerability is triggered, allowing the symbolic detector to focus on the same vulnerability rather than any arbitrary vulnerability. We use call stack guidance to constrain the exploration of a target version so that it only explores the basic blocks that can potentially lead to the same stack trace. Correspondingly, we translate call stack guidance into basic-block-level priorities to guide the exploration.

- Highest Priority: basic blocks that dominate the basic blocks in the call stack will receive the highest priority. This indicates that their execution is essential for reaching the bug while maintaining the same call stack. The set of such basic blocks can be identified through the dominator analysis on the control flow graph of functions in the call stack.
- Lowest Priority: basic blocks, upon the execution of which can cause deviations from the call stack, will receive the lowest priority. Consequently, a symbolic detector should avoid executing any of these blocks. They can be identified through reachability analysis.

In addition to the call stack guidance, we will need more fine-grained guidance if there are still too many possible execution paths that follow the same call stack. Specifically we propose to prioritize the execution path directly at the basic block level. The idea is that a basic block in the target version is likely to be non-critical if (1) the basic block is not executed in the reference version and (2) it remains unchanged in both the reference and target versions of the program. Therefore, the symbolic execution should prioritize the

exploration of branches whose basic blocks have higher priority. Note that when there are conflicts, path guidance must yield to call stack guidance because the most critical goal is to ensure the vulnerable function being reached. We translate path guidance into the basic-block-level priorities as follows:

- High Priority: basic blocks covered by the execution trace in the reference version of the program will receive high priority (lower than the highest priority).
- Low Priority: basic block not covered by the execution trace in the reference will receive low priority.

The indirect call guidance records the callee functions associated with each indirect function call encountered in the execution trace. Its primary role is to facilitate the accurate resolution of indirect function calls during the symbolic execution process, particularly in the target versions of the software under analysis.

3.4.2 Guidance Transformer

To enhance both the efficiency of symbolic execution and the precision of vulnerability detection in target program versions, it is essential to translate the three categories of guidance collected from a reference version. Specifically, one fundamental task is to map basic blocks from the binary form in the reference version, where execution traces are collected, to the LLVM IR in the target version, where symbolic execution is executed. One potential solution is first to map the binary-level basic blocks from reference to target. However, due to compiler optimizations, even if the source code lines are identical, their binary basic blocks may differ, making this solution undesirable.

Our solution employs source code as an intermediate representation to improve the mapping accuracy between the reference and target versions. The transformation sequence for basic blocks begins with the binary form in the reference version, moves to its source code, transitions to the source code in the target version, and ends in the LLVM IR of the target version. To facilitate these mappings, we use the debug information to transition between binary and source code and between source code and LLVM IR. Additionally, Git is employed for source code mapping between the reference and target versions. During the transformation sequence, we take care of multiple corner cases to enhance the precision (more details in §3.5.1).

After transforming the call stack to the target version, we verify the presence of the target line triggering the vulnerability. If absent, the target version is deemed non-vulnerable. If present, we examine whether there is a potential path from the entry to the target function in the call graph. A missing path directly results in a negative outcome.

- Medium Priority: basic blocks unique to the target version, which do not map to any basic blocks in the reference version, will receive medium priority. Compared to the low priority basic blocks – the ones seen in reference version yet not exercised, we are less certain about the utility of such basic blocks; therefore we prefer to explore them with a higher priority compared to the basic blocks that were seen in both reference and target versions but not exercised in the reference.

Then, all the guidance (call stack, five lists of varying priorities, and indirect call mapping) are forwarded to the subsequent component.

3.4.3 Symbolic Detector

After generating guidance for the target version of the program under analysis, the symbolic detector conducts under-constrained symbolic execution on these targeted versions. Specifically, the detector monitors all variables within the program to identify potential vulnerability patterns, such as use-after-free or out-of-bound access errors. We propose multiple improvements to enhance the ability of under-constrained symbolic execution, including but not limited to handling symbolic addresses, and symbolic sizes of allocated memory. The details are described in §3.5.2. Throughout this execution, the symbolic detector utilizes the guidance generated in prior stages to enhance its effectiveness.

Call Stack Guidance. Symbolic execution is initiated at a selected entry function, determined by examining the call stack. Specifically, execution starts at the first meaningful function in the call stack — we choose to start at the syscall handler [103] (which is typically several layers behind the generic syscall entry). The symbolic execution process ends upon detecting a vulnerability (resulting in a positive output) or upon hitting a time constraint (yielding a negative output). Importantly, the detector only checks for vulnerabilities upon reaching the specified target line in the guidance, avoiding hitting any unrelated bugs accidentally. Also, the basic blocks with the lowest priority are prohibited from execution.

Path guidance. When symbolic execution encounters a symbolic condition, it forks to explore both true and false branches. This forking behavior primarily contributes to the path explosion in symbolic execution. The path guidance is employed to address this. This

approach prioritizes exploration by first traversing branches with higher priority. When two branches have the same priority, one is randomly selected to be explored first.

Indirect Call Guidance. During symbolic execution, if we observe the indirect call target being assigned explicitly to a function pointer, we can unambiguously determine the indirect call target. Otherwise, we initially refer to the indirect call guidance to identify the indirect call target. If we find a match for the specific indirect call, we use the specific target from the guidance directly. Otherwise, we utilize the state-of-the-art type-based analysis [137] to resolve indirect calls (which may produce multiple targets).

3.5 Implementation

In total, the implementation of SymBisect has 4,726 LoC Python code for the Guidance Generator and Guidance Transformer and 4,347 LoC of C++ for the Symbolic Detector atop KLEE [72]. In the following sections, we will delve into further details regarding the Guidance Transformer (§3.4.2), and Symbolic Detector (§3.4.3).

3.5.1 Guidance Transformer

Code formatting. Because we employ source code as an intermediate representation during the guidance transformer, we require each source code line to be associated with only a single basic block. To achieve this, we develop a simple source code formatter that divides composite lines into simpler ones. For instance, splitting "} else if(cond){" into two distinct lines. This is done for both the reference version and the target version at the beginning.

Accurate coverage collector. SymBisect leverage KCOV mechanism to discern which sections of the code have been covered. Syzkaller offers a tool to save the coverage data from KCOV. However, this operation is not always reliable. When the kernel crashes, some coverage can be lost. To improve this, SymBisect modifies the kernel to record the KCOV buffer in a log upon a kernel crash.

Refine guidance. compiler optimizations (compiling the Linux kernel with -O0 is generally not supported), e.g., function inlining, and reordering, can lead to inaccuracies when mapping basic blocks in binary instructions into their corresponding source code lines with debug information – we find that the coverage of many basic blocks can be lost. To mitigate such impact, we implement an analysis of the basic blacks with the control flow graph and the dominator tree. We recover potentially lost basic block coverage under the following two conditions: 1) Should a line within a BB be marked as covered by a test case, it is necessary to mark all lines within that same BB as covered as well. 2) In instances where a covered BB is dominated by another BB (indicating that it is invariably executed after the dominating BB), it's essential that the dominator BB is also marked as covered.

3.5.2 Symbolic Detector

Under constraint symbolic variables. We choose to symbolize all variables without concrete values in static environments, including global variables and arguments of system calls. This approach allows us to explore a broader range of potential execution paths during our analysis.

Symbolic address. In its original form, KLEE does not adequately support

under-constrained symbolic addresses. When it encounters read/write operations to a symbolic address, KLEE typically generates a specific concrete address based on the current constraints.

The logic KLEE employs for dealing with under-constrained symbolic addresses is not reliable, particularly when faced with a multitude of such addresses. There might be instances where a symbolic address does not map onto any existing object. In such cases, arbitrarily concretizing this address to an existing object and proceeding with read/write operations can lead to incorrect outcomes.

Instead, we apply an improved mechanism in UCKLEE [78] to deal with symbolic addresses that have not been encountered before. When attempting to write/read to such a novel symbolic address, our system allocates a new object. Besides that, we maintain mappings between symbolic and concrete addresses. Therefore, subsequent attempts to access the same symbolic address will, in reality, be directed toward the corresponding concrete object as per the mapping. This procedure ensures that each symbolic address is consistently linked to a unique concrete object, thereby improving the precision of read/write operations and overall analysis.

Symbolic size. The original way KLEE allocates a new object with symbolic size is also not suitable for our situation. Specifically, if the size is symbolic, it generates a specific concrete size, and then KLEE tries to half its size until the size is no larger than a small constant (i.e., 128 in KLEE v2.2).

In our under-constraint cases, it will result in many objects with small sizes, such automatic concretization may result in the inaccuracy of the results. For example, if there

is a path that can only be explored with a size larger than the constant, it will always be skipped.

Instead, we implement a solution similar to the previous work[178] to handle this issue. We choose to track the symbolic sizes. We allocate the object with a large constant size in memory to make sure that the intended access to the object won't be missed and log the symbolic size. When there is a check against the size of an object, we always use the symbolic size.

Function modeling. To improve the scalability of symbolic execution, we manually model more general library functions belonging, such as strcpy(), malloc().

Vulnerability checker. The under-constraint nature of our symbolic execution will introduce some false positives when asserting the presence of vulnerability logic. To mitigate the problem, we concentrate on detecting the vulnerability on the corresponding line (called target line) in the target version where the vulnerability is triggered – we require the same line to be present in the reference and target version.

Once reaching the target line, for each read/write operation, we extract the address (usually symbolic) and find the corresponding object. If no corresponding object is found (usually happens in UAF cases), instead of allocating new under-constrained memory, we report the vulnerability directly. Otherwise, SymBisect compares the offset with the size of the object under current constraints. If the offset can be larger than the size (usually happens in OOB cases), SymBisect reports the vulnerability and terminates the execution. Finally, if none of these is detected, SymBisect keeps exploring various execution paths until a time limit is reached or runs out of paths to explore, leading to a negative result.

Tools	TP	FP	TN	FN	Accuracy	Precision	Recall	F-1 Score
SymBisect	237	29	348	31	90.7%	89.1%	88.4%	88.7%
Syzbot(PoC)	146	27	350	122	76.9%	84.4%	54.5%	66.2%
V0Finder	138	0	377	130	79.8%	100.0%	51.5%	68.0%
VSZZ	250	145	232	18	74.7%	63.4%	93.3%	75.4%

Table 3.1: **The results of vulnerable versions detection**

Tools	Correct	Incorrect	Accuracy
SymBisect	24	8	75%
Syzbot	16	16	50%
V0Finder	11	21	34.375%
VSZZ	18	14	56.25%

Table 3.2: **The results of bug-inducing commit identification**

3.6 Evaluation

In this section, we evaluate SymBisect based on the following three research questions.

- RQ1: How precisely does SymBisect identify the vulnerable versions for a specific vulnerability? How precisely does it determine the exact bug-inducing commit?
- RQ3: How effective is SymBisect, when compared with state-of-the-art (PoC-based/patch-based) bug-inducing commit identification methods?
- RQ4: How efficient is SymBisect in conducting its analysis? Specifically, how does the provided guidance/exploration strategy improve efficiency?

Evaluation Target and Vulnerability Dataset. We assess SymBisect on Linux kernel bugs reported on syzbot [97]. This choice is made due to several factors. First, syzbot is among the earliest and most mature continuous fuzzing platforms and the Linux kernel

is among the most popular open source software. Second, the Linux kernel is the largest software that is being continuously fuzzed today. Third, there are a variety and a large number of bugs reported on syzbot continuously, which require bisection. We believe our solution generalizes beyond the Linux kernel as it is likely more complex than most other software.

Therefore, we randomly sampled 50 bugs from syzbot reports that meet the following requirements: 1) reported in the last 4 years. 2) labeled to have OOB or UAF impact. 3) not race conditions (which our symbolic detector currently does not support). 4) with PoCs and the bugs can be reproduced. 5) the corresponding patch has a "Fixes:" tag (to be explained below).

A "Fixes:" tag is included in a patch that points to one or more previous commits that are considered to introduce the corresponding bug. We treat it as the ground truth because we verified that they are consistent with our definition of bug-inducing commits (see later for "ground truth verification"). Note that SymBisect does not require the presence of a "Fixes:" tag to operate; we choose such bugs to merely simplify the evaluation process.

For each vulnerability, our tool begins with the released vulnerable version and inspects every major release version (e.g., v5.10) until the oldest version, v4.20, in our dataset. Versions prior to v4.20 present compatibility issues with the Clang/LLVM toolchain. While more engineering work might address this, it diverts from our primary focus. . In total, our dataset consists of 645 bug-version pairs. We will determine whether each version is affected by a bug (vulnerable vs. non-vulnerable). We evaluate the accuracy of SymBisect against these bug-version pairs. Subsequently, to evaluate

bug-inducing commit identification, we retained the bugs introduced after v4.20 (32 in total): SymBisect employs a binary search to pinpoint the exact bug-inducing commit.

All experiments are conducted in Ubuntu-20.04 with 1TB memory and Intel(R) Xeon(R) Gold 6248 20 Core CPU @ 2.50GHz * 2. For each bug-version pair, we allocate a single CPU core for a maximum of 10,000 seconds of symbolic execution.

Comparison Targets. We compare SymBisect with the three following lines of work:

- PoC-based bisection. Syzbot bisects bugs with PoCs to find the commit that introduced the bug [25]. We employ a crawler to directly retrieve results from the website. In instances where bisection results are unavailable, we execute the PoC on the target kernels to get the results.
- Patch-based bisection with SZZ algorithm. As described in §2.2, this line of research assesses vulnerability-(un)affected versions by locating the vulnerability-introducing commit with SZZ and its variants. In this line of work, VSZZ [58] is the state-of-the-art tool and it's open source. We set up VSZZ with their default options according to the tutorials[30].
- Patch-based bisection with vulnerable code clone detection. These methods are based on code similarity comparison. V0Finder [194] is a recent vulnerable code clone detector that is used to discover the first version where a vulnerability is introduced. We set up V0Finder with their default options according to the tutorials[28].

Evaluation metrics. For the evaluation of determining the vulnerable versions for a specific vulnerability, for each bug-version pair, we will get a verdict as true positives (TP), true negatives (TN), false positives (FP), or false negatives (FN). Then we calculate the

corresponding accuracy, precision, recall, and F1 score. For pinpointing the precise bug-inducing commit, we received a binary result (either identifying the correct bug-inducing commit or not) from which we calculated the accuracy.

.

3.6.1 Accuracy of SymBisect (RQ1)

Accuracy of vulnerable version detection. As shown in Table 4.2, SymBisect achieves an overall accuracy of 90.7% over 645 versions, higher than all existing tools. Note that this evaluation is performed on a per-bug-version-pair basis.

Accuracy of bug-inducing commit identification. Table 4.1 shows the results of bug-inducing commit identification, SymBisect outperformed all the other cases with an accuracy of 75%. The reason the accuracy is lower (than vulnerable version detection) is that it aggregates the results from all kernel versions for a single bug. For example, if the vulnerability was introduced in v5.3, we might correctly label v5.4 as vulnerable; however, if we mistakenly labeled v5.3 as non-vulnerable, then we still will end up with an incorrect bisection result for the specific bug (FN). Upon manual inspection, we discovered that among these eight cases of inaccuracy, five were due to FPs, and three resulted from FNs.

False positives in SymBisect. SymBisect has 29 false positives (misidentifying non-vulnerable versions as vulnerable). The FPs generated by SymBisect tool arise from the intrinsic characteristics of under-constrained symbolic executions. For example, global variables are symbolized in our approaches, allowing the constraints to represent them as potentially holding any value of the specified type. However, such a variable could be

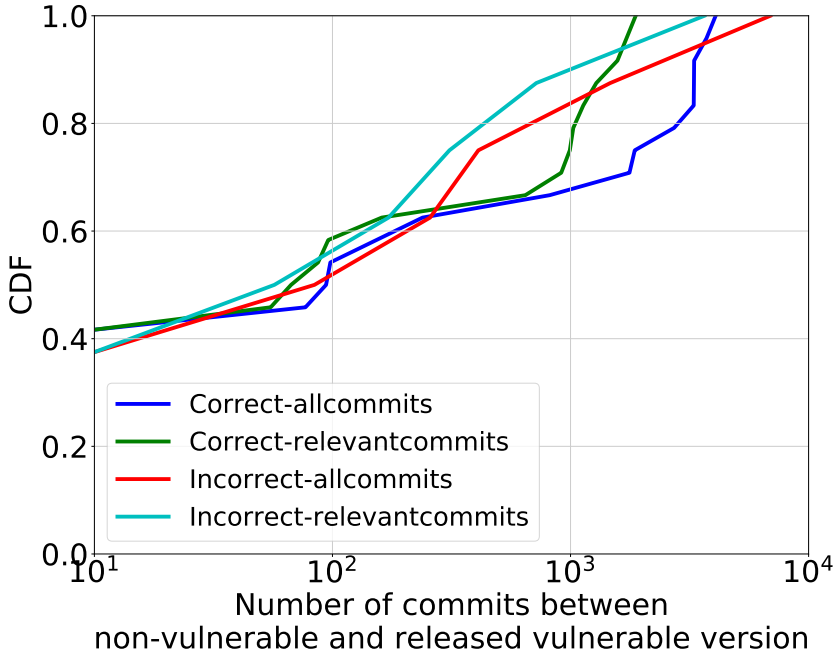

Figure 3.4: Comparison of commit number between correct and incorrect cases

hard-coded somewhere that symbolic execution cannot access. Consequently, such under-constraining can lead to SymBisect concluding infeasible behaviors in practice.

As an example false positive, we find an OOB bug that arises from a lack of checks against socket types. In the kernel, different types of sockets possess different sizes. The mappings between the socket type and the corresponding structure sizes are stored as global variables in the kernel, which are symbolized in our detectors. When under-constrained, the symbolized mapping can produce any sizes from a given socket type, leading to false positives.

False negatives in SymBisect. Our evaluation records 31 false negatives (misidentifying vulnerable versions as non-vulnerable). The primary cause of FNs is the scalability issue.

Certain vulnerabilities can be triggered only via a specific path, which might not be covered in the symbolic execution due to the time threshold, despite our effort to apply principled guidance during exploration. Moreover, the guidance may not be complete due to differences between the reference and target versions. If the symbolic execution lacks accurate guidance, it is likely to encounter scalability issues due to the complexity of kernels.

An example of this challenge occurs when a vulnerability site is influenced by a check against a pivotal variable. The vulnerability can be triggered only when this variable is set to a particular value in preceding functions. Yet, the distance between this value assignment and the condition check is substantial, with many functions with many state forks interspersed. Even with our guidance, satisfying such a nuanced condition in a limited time to activate the vulnerability proves challenging, resulting in false negatives.

3.6.2 Comparison (RQ3)

As shown in Table 4.2, SymBisect outperforms other tools effectively. It achieves higher accuracy (90.7% compared to the 77.1% average of preceding tools) and higher F1 scores (88.7% as opposed to 69.8%) than all previous tools. As expected, we observed that the main reasons for inaccuracies in existing PoC-based methods are the broken dynamical environment, inadvertent triggering of unrelated bugs, and evolving bug-triggering conditions as the code progresses. The failures of patched-based tools are due to their dependence on unreliable syntactic information and only consider a limited portion of bug-related code. In comparison, our solution based on static symbolic reasoning aims to capture the logic of the specific vulnerability and extend its scrutiny to

Reason	FN	FP	Solved in SymBisect
Hard to reproduce	38	0	15
Detector not introduced	8	0	8
Build/boot errors	14	0	14
Config disabled	9	0	9
Trigger another bug	0	27	27
Over-constraint on inputs	53	0	53
Total	149		126

Table 3.3: **The reasons of PoC-based method failed**

a much broader context beyond the confines of the patched function.

Improvements over syzbot bisection. Table 3.3 outlines the reasons for the PoC-based method's failures in our dataset. The first five types are cited from the official syzbot documentation[25], while the final reason, "over-constraint" is a reason we observed. In fact, we find that it is the most common reason for inaccuracies. Notably, SymBisect has effectively addressed 83% of the inaccuracies associated with the PoC-based approach. We will now detail the causes of each failure and how SymBisect addressed them, as follows:

- Vulnerability with low probability of triggering. PoC-based approach often struggles to reproduce bugs that have a very low probability of triggering even in the released version that corresponds to the PoC. At present, for every target version, syzbot conducts testing only 10 times [25]. It is probable that vulnerabilities may not be triggered within these limited attempts. The under-constraint feature of SymBisect enhances its capability to fulfill the preconditions necessary for triggering the bug. As a result, SymBisect yields accurate results for 15 of the 38 cases within the given time threshold.

- Detector not introduced. The PoC-based approach is dependent on specific detectors, like the KASAN sanitizer. Until these detectors are integrated into the kernel, PoCs cannot detect vulnerabilities effectively. In contrast, SymBisect is equipped with its own symbolic execution detector, eliminating the need for reliance on sanitizers in the Linux kernel.
- Build/boot errors. As we discussed in §3.2, the static feature of SymBisect bypasses the problem resulting from kernel boot errors.
- Config disabled. As PoC-bisection goes back in time, certain kernel configs may be forcefully disabled when they conflict with the other config options. In contrast, since our solution does not require the compilation of the entire kernel, we can simply force other config conflicts to be disabled and make sure the vulnerable modules involved are compiled into LLVM bitcode for our analysis.
- Accidental triggering of unrelated bugs. The PoC has the potential to activate unrelated kernel bugs that break the program. Current syzbot does not look at the exact crash, nor does it attempt to distinguish between different types of crashes, leading to some FPs. In contrast, our tool focuses on the specific bug only upon reaching the target line (and analyze its associated operations). This allows us to effectively sidestep this issue.
- Over-constraint on inputs. This is essentially due to changes in the underlying bug-triggering conditions. Executing the original PoC does not always activate the bug in some vulnerable versions. Input mutations become necessary under these circumstances. The under-constrained symbolic execution approach treats all potential entry function arguments and global values comprehensively, effectively addressing these false negatives.

Figure 3.5 presents an OOB vulnerability. Specifically, in function `mpol_parse_str()` if the str variable starts with "=", the flags variable will reference the first byte of str. If a certain condition at line 2 is met, the program skips to line 4. Here, a write operation occurs that exceeds the boundaries of str, leading to an out-of-bounds write. The PoC-based syzbot bisection incorrectly pinpoints a bug-inducing commit which modified the function `shmem_parse_one`—the caller of the `mpol_parse_str()` function. Prior to this misidentified commit, another check at line 8 was in place against the opt variable. The initial PoC fails this check, causing syzbot to label versions before this commit as non-vulnerable. However, by using a different input that bypasses this check, the bug remains exploitable. Instead, SymBisect symbolizes the inputs, making it easier to bypass such checks as long as a feasible solution exist.

Improvements over V0Finder. V0Finder failed to discover 107 vulnerable versions out of 230 cases, resulting in a low recall of 46.5%. The main reason is that V0Finder does a strict syntactic similarity comparison for the whole function. Specifically, after normalization and abstraction, it concludes that the target version is vulnerable only if the patch functions are strictly the same as those in the released version. Thus it cannot detect the vulnerable cases that are syntactically different, but convey the same vulnerable functionality.

In Figure 3.6, we see an illustrative example. Here, a 4-byte size variable is prone to an overflow at line 3. To address this, the patch modifies the variable's size to 8 bytes. However, the bug-inducing commit pinpointed by V0Finder is actually a feature

The vulnerable function:

```
int mpol_parse_str(char *str,...)
1   char *flags = strchr(str, '=');
2   if(condition)
3       goto out
    ......
4   if (flags)
5       *flags++ = '\0';
    ......
out:
6   if (flags)
7       *--flags = '=';
```

The incorrect Bug-inducing Commit (Identified by Syzbot Bisection):

```
static int shmem_parse_one(...)
    ......
8 - else if (!strcmp(opt, "mpol")) {
  -     ......
9 -     if (mpol_parse_str(value, &ctx->mpol))
        ......
10+ if (IS_ENABLED(CONFIG_NUMA)) {
  +     ......
11+     if (mpol_parse_str(param->string, &ctx->mpol))
```

Figure 3.5: Case study of syzbot FN

enhancement commit, unrelated to the vulnerability. This commit introduces multiple lines into the patched function. Due to this, V0Finder incorrectly designates all preceding versions as non-vulnerable, leading to a multitude of false negatives.

SymBisect, instead of syntactic comparison, extracts accurate semantic information. Thus it can distinguish vulnerability-irrelevant changes from significant changes effectively. Furthermore, it does not rely on patches. Whether the patch changes a function or not is irrelevant to SymBisect. As a result, SymBisect can eliminate a

The Patch:

```
int qrtr_endpoint_post(...)
    struct qrtr_cb *cb;
1 - unsigned int size;
2 + size_t size;
    ......
3   if (len != ALIGN(size, 4) + hdrlen)
    goto err;
```

The incorrect Bug-inducing Commit
(Identified by V0Finder):

```
int qrtr_endpoint_post(...)
    ......
+   if (cb->type == QRTR_TYPE_NEW_SERVER) {
+       const struct qrtr_ctrl_pkt *pkt = data + hdrlen;
+       qrtr_node_assign(node, le32_to_cpu(pkt->server.node));
+   }
```

Figure 3.6: Case study of V0Finder FN

large number of FN cases of V0Finder. This significant advantage is largely due to the differing foundational design principles of the two systems.

Improvements over VSZZ. VSZZ processes a patch as input and identifies the vulnerability-introducing commit by backtracing the patch's deleted lines through the code commit history to the earliest instance, facilitated by line matching. The earliest commit where these deleted lines were initialized is then marked as the commit that induced the bug. When multiple deleted lines originate from different commits, VSZZ selects the earliest of those commits as the bug-inducing commit. If the patch does not have any deleted lines, VSZZ identifies the commit that initialized the file mentioned in the patch as the bug-inducing commit.

Figure 3.7 illustrates a typical scenario where the underlying assumption fails,

The Patch:

```
int squashfs_read_data(...)
1 - TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
2 + TRACE("Block @ 0x%llx, %scompressed size %d\n", index - 2,
        compressed ? "" : "un", length);
      }
3 + if (length < 0 || length > output->length ||
  +          (index + length) > msblk->bytes_used) {
4 +     res = -EIO;
5 +     goto out;
  +  }
```

The incorrect Bug-inducing Commit :

```
    ...... (initialize the file)
6 +  TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
```

The correct Bug-inducing Commit :

```
int squashfs_read_data(...)
    ......
7   TRACE("Block @ 0x%llx, %scompressed size %d\n", index,
  -         compressed ? "" : "un", length);
8 - if (length < 0 || length > output->length ||
  -         (index + length) > msblk->bytes_used)
9 -     goto block_release;
```

Figure 3.7: Case study of VSZZ FP

leading to a false positive. The deleted line 1 in the patch function is not created by the vulnerability-inducing commit, leading to backtracing to an earlier point. All commits situated between the commit identified by VSZZ and the actual inducing commit will be marked as FPs. In detail, the vulnerability was brought into the codebase by a commit that removed a certain validation check at line 8, then the vulnerability was patched by putting the check back in. However, the line they removed from the patch was just for logging that is not really related to the vulnerability. VSZZ traced this logging line back to when the whole function was first added, resulting in some FPs.

Strategy	Implementation
SymBisect	Exploration + Indirect call + Stack + Path
Pure Exploration	Exploration + Indirect call
Pure Re-tracing	Indirect call + Stack + Path
Stack	Exploration + Indirect call + Stack
Path	Exploration + Indirect call + Path

Table 3.4: **The relationship between strategy and guidance**

Basically, the commit that introduces the vulnerability may not alter the patch function at all, as demonstrated in our motivating example. Even if it does alter the patch function, it may not modify the deleted lines in the patch, just as in the above example. Furthermore, even if the bug-introducing commit does change the deleted lines, it may only modify them rather than create them. In such cases, VSZZ may backtrace beyond the actual bug-introducing commit. These factors contribute to 112 false positives, a significantly higher figure than those seen with the other methods.

In contrast, our semantic method does not hinge on such a strong assumption. The symbolic execution engine accurately extracts semantic information, clarifying their relationship with the vulnerability.

3.6.3 Scalability of Different Exploration Strategies (RQ4)

To understand how the guidance helps with the overall results, we conduct a comparative study against alternative strategies. Fundamentally, SymBisect balances the exploration (i.e., allowing execution of the basic blocks in the medium-priority list) with re-tracing (i.e., aligning the execution trace with the one in the reference version). Therefore, we consider the following strategies that fall under various places in the

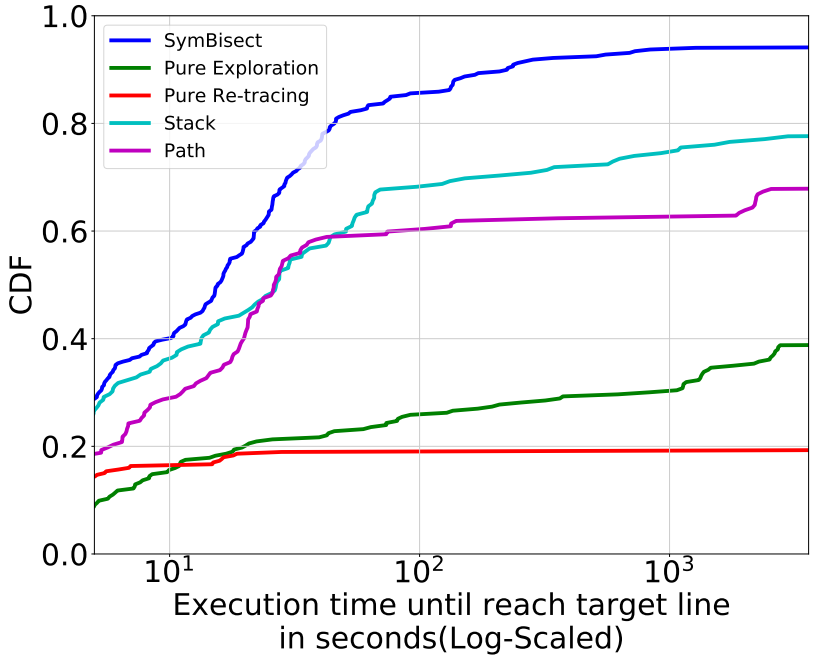

Figure 3.8: Scalability Evaluation

spectrum: (1) pure exploration without any re-tracing or guidance (no consideration of basic block priorities), (2) pure re-tracing strictly following path guidance (i.e., when a branch leads to a high/highest priority exists, the other branches are prohibited from execution), (3) exploration with call stack guidance only. (4) exploration with path guidance only.

3.7 Discussion

Exploration range. As discussed in §3.2, Relying solely on patch functions presents inherent disadvantages, prompting us to explore entire traces in order to gather

comprehensive information relevant to vulnerabilities within the program. However, these traces may encompass thousands of functions, with the majority of them unrelated to the vulnerability at hand. Consequently, achieving a balance between scalability and accuracy primarily relies on determining the appropriate exploration range. While we employ specific heuristics to limit the range, there is still room for a more systematic approach to this decision-making process. For example, we envision one can apply static analysis (less precise but more scalable) to identify the vulnerability-related functions in advance, then skipping the unrelated functions when applying symbolic execution. Developing such a solution would significantly improve our capability to identify and address vulnerabilities without overwhelming our resources.

Support more bug types. The types of vulnerabilities supported by SymBisect depend on the symbolic engine it is based on (currently KLEE) and the detectors built on top of it (or provided by KLEE itself). SymBisect currently supports bugs that manifest as OOB and UAF, including type confusion and integer overflow bugs that manifest as OOB. There are a few types of bugs that are interesting to support for future improvements: (1) additional bug types such as use-before-initialization [204], (2) bugs that require precise reasoning across multiple syscalls, and (3) race conditions bugs. For (1), it requires additional symbolic detectors to recognize other bug types. For (2), symbolic execution across multiple syscalls is feasible but presents an additional scalability challenge. For example, in some OOB cases, the allocation and use of the vulnerable object occur in different system calls. Without analyzing the allocation, the analysis of the subsequent syscall on use will be under-constrained and therefore potentially lead to

false positives. This means we will need to first collect the symbolic expression for the object size (in one syscall), and then reason about whether the use can go out-of-bounds (in another syscall). We envision an optimization to terminate the symbolic execution of the allocation syscall earlier, as soon as the object size info is collected and leave other unexplored variables under-constrained. For (3), there are specialized symbolic detectors that can detect specific race condition bugs, e.g., multi-reads and double-fetch [202]. In the context of bisection, we envision that a more general approach is to recognize the interleaving points [203] and record the desired interleaving during the execution of the PoC in the reference version and use it to guide the execution of the target version.

Support bugs without PoCs. When a fuzzer discovers bugs, it usually generates a corresponding PoC, but there are exceptions. In some cases, syzakaller only produces a bug report. We wish to point out that our tool does not necessarily have to rely on PoCs. Instead, as long as we can obtain traces that trigger the vulnerability, it would be sufficient to guide the symbolic execution. For example, with hardware support (e.g., Intel Processor Trace [94]), we envision bug reports can be accompanied with corresponding control flow information.

3.8 Related Work

Under-constrained symbolic execution in OS kernels. UCKLEE [158] represents the initial implementation of an under-constrained symbolic execution virtual machine based on KLEE. It is primarily utilized for patch verification as well as rule-based generalized checks, encompassing areas such as memory leaks, uninitialized data, and user input vulnerabilities.

UBITect [204] and IncreLux [205] utilize under-constrained symbolic execution to identify feasible paths and mitigate false positives in static analysis when detecting Use-Before-Initialization (UBI) bugs. SID [195] aims to distinguish security-related patches from other bug fixes, which is different from our work. It attempts to set up a model for several types of vulnerabilities with the help of under-constraint symbolic execution, rather than simply extracting and comparing characteristics. Besides, previous studies that attempted to perform symbolic execution on operating system kernels addressed the scalability issues using the following methods: 1) Decrease the scope of symbolic execution when analyzing operating system kernels. For example, performing intra-procedural analysis on a specific function such as the patch function [207] [195]. However, the approach may not be suitable for our purposes. The existence of a vulnerability is not determined by a single function. 2) Concretizing symbolic inputs and global variables [67, 197]. In our cases, it will result in an over-constraint problem.

Dynamic vulnerable version identification. The information about the affected versions of a vulnerability is quite important [167]. Dai et al. [75] proposed a PoC migration approach that takes a PoC as input and migrates the PoC to verify other affected versions. However, it specifically targets user-space programs. Furthermore, as demonstrated in § 3.6, over-constraint on inputs is only one of the causes of failure.

Information-retrieval-based bisection. Locus [188] was the initial method to pinpoint bugs at the software change level using token similarities from bug reports. ChangeLocator [196] determines Bug-Inducing Commit (BIC) using crash call stack information. Orca

[61] ranks commits based on bug symptoms, like exception messages or customer feedback. Bug2Commit [142] aggregates features from bug reports and commit, averaging their vector representations. FONTE [52] identifies BIC via test coverage. It ranks commits by the suspiciousness of their modifications. Despite their scalability, these methods fall short in accuracy. As mentioned in the Background section, The state-of-the-art, Fonte, only reaches a 36% accuracy rate.

3.9 Conclusion

The identification of vulnerable versions of Open Source Software and pinpointing bug-inducing commits are crucial for vulnerabilities uncovered through fuzzing. In response to this, we introduce SymBisect, a precise methodology grounded in symbolic analysis. The central principle is that detailed symbolic information tends to be more stable compared to both the original PoC and syntactic similarity assessments during software evolution. Our experimental results confirm that SymBisect not only significantly surpasses the existing PoC-based approach in terms of accuracy, but also outperforms methods that rely on patches. With the insights gained from SymBisect about vulnerable versions, developers can precisely locate the bug-inducing commit. This empowers them to address the potential threats brought about by fuzzing vulnerabilities, thus promoting a more secure software ecosystem.

Acknowledgment

We thank anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation under Grant #2155213, #2247881 and a Google Gift.

Chapter 4

Breaking Barriers: Accurate Bug Bisection with Full Patch Context and LLM Insight

Abstract

Bug bisection has been an important security task that aims to understand the ranges of software versions impacted by the bug, i.e., identifying the commit that introduced the bug. However, traditional patch-based bisection methods are faced with several significant barriers: For example, they assume that the bug-inducing commit (BIC) and the patch commit modify the same functions, which is not always true; they often rely purely on code changes, while the commit message frequently contains a wealth of vulnerability-related information; they are also based on simple heuristics (e.g., assuming the BIC initializes lines deleted in the patch) and lack a logical analysis of the

vulnerability.

In this paper, we make the observation that Large Language Models (LLMs) are well positioned to break the barriers of existing solutions, e.g., comprehend both textual data and code well in patches and commits. We develop a comprehensive multi-stage pipeline leveraging LLMs to (1) take advantage of full patch information, (2) have LLM assess logic of the bug and the likelihood of a commit being the one that introduced the bug, and (3) gradually narrow down the candidate with multiple down-select processes. In our evaluation, we demonstrate that our approach achieves significantly better accuracy than the state-of-the-art solution by more than 38%.

4.1 Introduction

N-day vulnerabilities are known security flaws which are often not fixed timely, due to complex dependency chains and limited maintenance resources [85]. The widespread reuse of open-source projects exacerbates this problem, as there are usually multiple downstream distributions maintained by different parties in the ecosystem [211], making it difficult to apply upstream security patches timely across all distributions. Research has shown the popularity and severity of this problem in critical open-source projects such as Linux [132] and Android [211], potentially affecting billions of users.

The information of affected software versions of a specific vulnerability is crucial for N-day vulnerability mitigation. To obtain such information, it is necessary to locate the commit introducing the vulnerability (*i.e.,* bug-inducing commit, or *BIC*) — an essential task known as *bug bisection*. In this paper, we align with previous studies [210] and define a

BIC as a commit that introduces a software bug into a program. It is possible for multiple commits to contribute to the bug, with the final commit making the bug triggerable. In such cases, we consider the final commit as the *BIC*, as it marks the point when the vulnerability is considered to exist.

Automated bug bisection can significantly speed up the bug-fixing process in downstreams (*e.g.,* 2.23x on average for Google's codebase, according to a previous study [7]), however, achieving a high accuracy remains challenging. Consequently, public information of vulnerable versions (*e.g.,* in NVD database [19]) is usually incomplete or inaccurate, as shown in previous study [54, 198, 112].

Existing automatic bug bisection approaches can be classified into several categories, each with its own significant limitations:

(1) PoC-Based. Directly or symbolically execute the PoC (Proof-of-Concept) against each software version to test whether the vulnerability can be triggered. [97, 210] Though straightforward, this approach suffers from limited availability of vulnerability PoCs. Furthermore, direct PoC execution [97] often fails due to subtle variations across software versions, resulting in low accuracy [26], while symbolic analysis [210] is known to be expensive, and only supports limited bug types (*e.g.,* use-after-free, out-of-bounds memory access).

(2) Bug Report-Based. These approaches first collect available bug reports and then identify possible BICs by their "relevance to the bug" [52, 188, 61], with simplified assumptions such as "a BIC should touch the code where the failure happens". Similar to PoCs, detailed bug reports are often not available. Moreover, the simplified assumptions/heuristics may not

hold in reality, reducing the accuracy, *e.g.,* Fonte's [52] accuracy drops to 36% when N=1 in its top-N ranking algorithm.

(3) Patch-Based. Being the most widely used, these approaches statically analyze the bug-fix commit (usually available for vulnerabilities in open-source projects) to "infer" the bug-inducing commit in the commit history. Existing techniques in this category [170, 58, 123, 194, 212, 192] generally rely on manually developed, hardcoded, and thus inherently imprecise heuristics. For example, one common one is to treat the commit that introduces one or more lines deleted by the bug-fix as the bug-inducing commit; however, there are many situations where the bug-inducing commit and the fix commit do not intersect. For example, a patch can add an additional security check before the original vulnerable code (without removing any existing lines of code), potentially in a different function, or the deleted lines in the bug-fix are irrelevant. Another significant shortcoming is that existing approaches usually only analyze the structured code changes in the bug-fix patches, while unable to take any advantage of the commit messages in the unstructured natural language form. However, commit messages often contain rich and valuable information that can boost the bug bisection performance (*e.g.,* hints on the vulnerability root causes).

In this paper, we target patch-based bisection because it is the most widely applicable scenario — not all bugs come with PoCs or crash reports. We specifically have three goals: i) support all types of patches and vulnerabilities, ii) utilize full patch information including both code changes and commit messages, and iii) go beyond the simple hardcoded heuristics and make accurate decisions based on analysis of the vulnerability logic. To achieve these goals, we propose SymBisect, an LLM-powered

highly accurate bug-bisection solution. Our core insight is that LLM is capable of understanding both code and natural languages, extracting useful information for bug-bisection. Recent LLM models (*e.g.,* OpenAI o1) also show impressive abilities in code reasoning.

Though promising, there are several obstacles to the direct application of LLMs. First, LLMs tend to produce *excessive false positives*, aggressively and incorrectly labeling commits as bug-inducing. Second, they can exhibit *self-consistency* issues, yielding conflicting decisions across multiple runs. Finally, the *cost* of using LLMs on large-scale software is prohibitive: modern projects often contain tens of thousands of commits, and processing each one naively can lead to substantial token consumption and increased time and monetary costs.

To overcome these challenges, we design and implement a multi-step filtering approach. First, we perform coarse-grained filtering to extract potential BIC candidates at scale (§4.3.3), based on various patch information (*e.g.,* code changes, commit message keywords). This process is cheap and efficient as it rely minimally on the LLMs, yet, it effectively narrows down the BIC search scope for later more expensive stages, improving both accuracy and performance. Next, we conduct fine-grained BIC filtering, utilizing LLMs' code and natural language reasoning capability. To address the false positives, our filtering is multi-round, ensuring that the LLM is sufficiently exposed to all promising BICs for a better comparative assessment. This design significantly boosts our accuracy. Finally, we carefully adopt the majority voting mechanism in limited key steps, to mitigate LLM's self-consistency issues without incurring high performance overhead.

We extensively evaluate SymBisect on Linux kernel, one of the most complex and important open-source software. The results show that SymBisect achieves a remarkable accuracy of 91%, significantly outperforming state-of-the-art bug-bisection approaches. We summarize our contributions as follows:

(1) We developed a novel and significantly different solution: an automatic bisection tool called SymBisect. It overcomes the limitations of previous tools by fully leveraging patch information, including commit messages, and utilizing advanced LLMs for logical analysis of code. We plan to open-source the solution to enable the reproduction of results and support further research.

(2) We proposed a new multi-step filtering approach to address the challenges of directly applying LLMs in bisection tasks, significantly improving the accuracy of SymBisect.

(3) We evaluated the performance of SymBisect against other state-of-the-art methods and demonstrated that it significantly outperforms existing tools (91% vs. 51%). We analyzed the barriers that previously hindered higher accuracy in earlier tools and explained how our solution overcomes these challenges.

4.2 Motivation

4.2.1 Motivating example

Figure 4.1 illustrates a race-condition-induced use-after-free vulnerability. However, in this example, the true BIC that introduced the vulnerability modified a completely different function from the one targeted by the patch. Specifically, the

The Patch:

ty: n_gsm: fix race condition in status line change on dead connections

gsm_cleanup_mux() cleans up the gsm by closing all DLCIs, stopping all timers, removing the virtual tty devices and clearing the data queues. This procedure, however, may cause subsequent changes of the virtual modem status lines of a DLCI. More data is being added the outgoing data queue and the **deleted kick timer** is restarted to handle this. At this point many resources have already been removed by the cleanup procedure. Thus, a kernel panic occurs. Fix this by proving in gsm_modem_update() that the cleanup procedure has not been started and the mux is still alive.

```
static int gsm_modem_update(...)
+   if (dlci->gsm->dead)
        return -EL2HLT;
```

The Bug-inducing Commit(partly):

```
static void __gsm_data_queue(...)
+   mod_timer(&gsm->kick_timer,...);

+static void gsm_kick_timer(...)

static int gsm_cleanup_mux(...)
    /* Finish outstanding timers, making sure
they are done */
+   del_timer_sync(&gsm->kick_timer);
```

Figure 4.1: Motivating example

bug-inducing commit introduced a new thread that runs the newly introduced function `gsm_kick_timer()`, while the **patch commit** adds a check in `gsm_modem_update()` to ensure that the cleanup process has not been initiated before it it allowed to create the

timer thread (it will eventually call `__gsm_data_queue()`), and hence eliminating the possibility of a race.

SymBisect, is the state-of-the-art PoC-based Bisection method, cannot accurately extract the BIC in this case because: 1. There is no existing Proof of Concept (PoC). 2. SymBisect only supports two specific types of bugs: Out-of-Bounds (OOB) and Use-After-Free (UAF). Specifically, it does not support race condition cases.

VSZZ, the state-of-the-art method in the SZZ family [170, 124, 74, 145, 58], fails to handle such cases because its fundamental assumption is that the BIC modifies the same functions as the patch (specifically, that the BIC initializes the lines deleted in the patch). However, in this case, the BIC and the patch modify completely different functions.

V0Finder, an advanced vulnerable code clone detection method, identifies vulnerable versions by comparing the patch functions of the target version and the patch functions before the patch, after normalization and abstraction. If they are identical, the target version is considered vulnerable. However, this method fails in the illustrated case because the BIC does not modify the patch functions at all.

This case motivates us to think of a better approach for extracting candidate commits, one that goes beyond merely tracking patch functions. In fact, in this example, although the BIC modifies a different function than the patch, we note that `gsm_cleanup_mux()` is explicitly mentioned in the patch description. This provides an important hint that we can expand our focus to not only analyze code changes but also extract valuable information from commit messages.

Figure 4.2 illustrates another motivating example, which represents a NULL

The Patch

```
thermal_zone_device_register_with_trips(...)
release_device:
    put_device(&tz->device);
-   tz = NULL;
remove_id:
    ida_free(&thermal_tz_ida, id);
free_tzp:
    kfree(tz->tzp);
```

The correct BIC

```
thermal_zone_device_register_with_trips(...)
remove_id:
    ida_free(&thermal_tz_ida, id);
+free_tzp:
+   kfree(tz->tzp);
```

The incorrect BIC (VSZZ)

```
thermal_zone_device_register_with_trips(...)
+release_device:
+   put_device(&tz->device);
+   tz = NULL;
```

The incorrect BIC (V0Finder)

```
thermal_zone_device_register_with_trips(...)
-  if (!ops) {
+  if (!ops || !ops->get_temp) {
      pr_err("Thermal zone device . . .");
```

Figure 4.2: Motivating example #2

pointer deference bug. In this example, although the BIC modifies the same function as the patch and is thus included among the candidates, the flawed heuristics of traditional methods prevent them from accurately identifying the correct BIC.

Specifically, the buggy code incorrectly sets `tz` to NULL under certain conditions, which causes the NULL pointer to be dereferenced subsequently in `kfree(tz->tzp)`. The patch fixes this vulnerability by removing the assignment that sets tz to NULL.

The commit introducing this vulnerability added a `kfree()` function call where the null dereference occurs. Before the BIC, the `kfree` function call did not exist, so naturally, the null dereference was not an issue.

VSZZ does not try to understand the logic of the vulnerability. Instead, it tracks the lines deleted in the patch, leading back to the commit that initialized the line (in this case, an earlier commit that first created the line of `tz = NULL`). This completely overlooks the actual BIC.

V0Finder's flawed heuristics, comparing hash values (essentially string matching after normalization and abstraction) of the whole patch function, take a different approach, focusing on all commits that modified the patched function. Specifically, it identifies a commit that modified the patch function (the latest one before the patch) as the BIC, but this modification is unrelated to the vulnerability. V0Finder does not determine whether the modification is logically connected to the vulnerability; it simply assumes that, before the modification, the function was different, and therefore, the vulnerability did not exist.

4.2.2 Limitations of previous methods

Based on the motivating examples, we summarize the key weaknesses in patch-based methods, including *SZZ algorithm and its variants* [170, 124, 74, 145, 58] and most vulnerable code clone detection solutions [123, 194, 212, 163]. They suffer from the following limitations:

1) They often only consider code changes, ignoring commit messages, which frequently contain crucial information about vulnerabilities.

2) They fail to account for cases where a Bug-Introducing Commits (BIC) does not change the functions affected by the patch.

3) Many of them focus on deleted lines in the patch, making them ineffective when patches only include added lines or when the deleted lines are not critical to the vulnerability.

4) They tend to treat all code changes (such as deleted lines) equally. In reality, not all changes are of equal significance to the vulnerability.

5) Their judgments are often based on simple heuristics rather than logical reasoning. For example, VSZZ, the state-of-the-art SZZ method, traces back commit history to the earliest commit (instead of the most recent) that introduces the deleted lines of a patch. Such heuristics are often not accurate.

4.2.3 Insights

Revisiting the motivating example, we propose three design goals for an improved solution:

1) Leverage Full Patch Context: The solution should utilize the complete patch context, including both the patch code diff and commit messages, as these provide critical clues about the bug-inducing commit.

2) Minimize Assumptions and Requirements: Unlike approaches such as VSZZ, the solution should support patches that only add lines. It should also handle all types of bugs rather than being restricted to specific categories (e.g., SymBisect). Additionally, it

must accommodate patches that do not modify functions, a limitation seen in V0Finder.

3) Incorporate Logical Reasoning: The approach should analyze the logic of the vulnerability to make a decision on the bug-inducing commit, rather than depend on simplistic and hardcoded heuristics like those used in VSZZ and V0Finder.

To achieve these goals, we propose leveraging large language models (LLMs) for the task of bug bisection. LLMs are well-suited for this purpose due to their ability to comprehend both code and patch descriptions. Moreover, they are trained on all types of bugs and patches and thus not limited to reasoning about specific types of bugs/patches. LLMs have demonstrated effectiveness in various bug analysis tasks [201, 181, 129, 180] and have been improving one generation after another.

4.3 Design

4.3.1 Design Motivation

Though LLMs show great potential in improving existing bug bisection techniques, it remains unclear how exactly they should be utilized to achieve the best performance. Intuitively, bug bisection is the process of identifying the BIC from a list of candidate commits related to a given patch and its associated vulnerability. Thus, we can divide the process into two steps: 1) extracting candidate commits from the commit history, and 2) selecting the exact bug-inducing commit from the candidate commits.

Below, we will discuss these two steps separately, starting with a baseline design. We then discuss the challenges we empirically encountered when testing such a design with a small set of real-world cases. The observations and insights gained when evaluating them

enable us to come up with corresponding solutions. Through this iterative refinement, we ultimately present our final, optimized design.

Collection of BIC candidates.

Baseline. A straightforward way of collecting BIC candidates is to generalize the state-of-the-art method, i.e., VSZZ and V0Finder, which considered only the commits that changed patch function(s). Specifically, we can collect *all historical commits that modify the patched function(s).* The intuition is that this represents a superset of commits encompassing the BICs identifiable by previous methods. It can also overcome their limitation of not supporting patches with added lines only (no deleted lines).

Challenge #1. The total number of commits that modified the patch function is often quite large in the commit history. Too many candidates can reduce the accuracy of the LLM (as observed in our preliminary experiments). Moreover, some BICs do not modify the patch functions at all, which will be missed by the above solution.

Observation #1. Not all functions or lines modified in the bug-fix commit are equally important or relevant to the vulnerability. For example, some code changes are merely for refactoring purposes without changing the semantics of the code. Previous methods also attempt to identify irrelevant code changes. However, their methods are limited to only simple patterns such as adding or removing comments [194, 123].

Solution #1. We change the simple non-distinguishing function-based candidate selection to a fine-grained critical-line-based selection. Specifically, we first identify the most relevant changed lines to the vulnerability from the bug-fix commit, with LLM's help, and then include only historical commits that touch these lines in the candidate list. This approach

significantly reduces the number of candidates to inspect (by 81% in our evaluation on average). An additional benefit is that this critical-line-based method enables us to include BIC candidates beyond the function scope (*e.g.,* changing a global variable definition).

Challenge #2. While this improves the accuracy if the BIC indeed modified the critical lines, it still does not solve an aforementioned problem — the code change made in the bug-inducing and bug-fix commits can be disjoint (*e.g.,* in different functions or files).

Observation #2. The patch commit messages often contain useful clues hinting at the vulnerability's root cause and connecting it to the bug-fix (*e.g.,* the commit message of a bug-fix in `foo()` may mention that the vulnerability originates from `bar()`). The motivating example illustrated this point.

Solution #2. Going beyond the function- and critical-line-based candidate selection, we can leverage LLMs to select additional BIC candidates using hints extracted from the commit messages (*e.g.,* commits that modified a function mentioned in the commit message). Because we look for functions or variables outside of the patched function, it is complementary to the previous two methods by design.

Selection of BIC from candidates

Baseline. Given that LLMs are fully capable of comprehending code and textual data, a baseline method can be: let LLMs inspect the BIC candidates in reverse chronological order, the first one recognized as BIC will be the selected one (*i.e.,* as mentioned in §4.1, we define BIC as the last commit contributing to the vulnerability). While plausible, we identified multiple challenges during our preliminary experiments.

Challenge #3. High false positive rate. During the reverse chronological traversal, LLM

tends to recognize BICs overly eagerly and thus stop early, missing the real BICs, resulting in low accuracy.

Observation #3. Despite having FPs, LLMs perform well in discerning the real BIC when it is presented together with FP candidates.

Solution #3. We adopt a two-round BIC selection: 1) let the LLM inspect *all* candidates and identify *all* potential BICs, without early termination, and 2) let the LLM *compare* all the identified BICs and select a final one.

Challenge #4. Though our evaluation shows that the critical-line-based method improves the overall accuracy compared to the function-based method, vulnerability-relevant lines can still sometimes be missed by LLMs in some cases, resulting in false negatives in BIC recognition.

Observation #4. We have three methods for generating BIC candidates, each with its own trade-offs. Their results can complement each other.

Solution #4. To avoid missing the correct BICs, we feed all three sets of BIC candidates to the LLM (using the three methods described earlier). To further improve accuracy, we make a final selection from the results generated by different methods (*e.g.,* function-based and critical-line-based), rather than merging the candidate commits at the beginning. This is because the accuracy of the LLM tends to drop when we feed a large set of BIC candidates. In other words, we will feed only three candidates to the LLM, when it is making the final verdict. In most cases, even if a method produces a wrong candidate, it will not be selected among the final three. However, if it does produce the correct candidate, it will be very likely chosen finally.

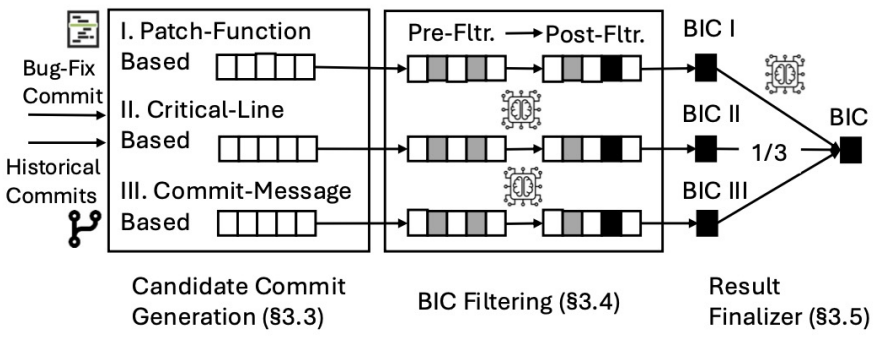

Figure 4.3: Workflow of SymBisect

4.3.2 Workflow

Motivated by the above design explorations, we present the workflow of our final design of SymBisect in Fig. 4.3. As we can see, there are three overall stages: 1) Candidate Commit Generation. 2) BIC Filtering. 3) Result Finalizer.

Candidate Commit Generation. Given a bug-fix commit, this stage's goal is to list all historical commits that could potentially be the BIC for future investigation (*i.e.,* candidate generation). As described previously, we have three candidate commit generation methods, based on patch functions, critical lines, and commit messages, respectively. These methods can complement each other (Solution #1 and Solution #2 in Section 4.3.1).

BIC Filtering. At this stage, we aim to select the most likely Bug-Inducing Commit (BIC) from each list generated in the first stage, resulting in up to three final BIC candidates. This process is divided into two phases: the pre-filtering phase, which identifies possible BICs, and the post-filtering phase, which selects the most likely BIC (Solution #3 in Section 4.3.1).

Result Finalization. At this stage, we finalize our decision by selecting one final Bug-

Inducing Commit (BIC) from the potential BICs (up to three) identified during the BIC filtering stage. (Solution #4 in Section 4.3.1)

Majority voting. LLMs sometimes make different decisions regarding BIC selection in multiple runs (*i.e.,* self-consistency). We observed that there usually exists a "dominating" decision occurring in most runs. Thus We adopt a "majority voting" mechanism in our design, where we run LLMs multiple times for BIC identification and select the most frequent answer as the final decision.

4.3.3 Candidate Commit Generation

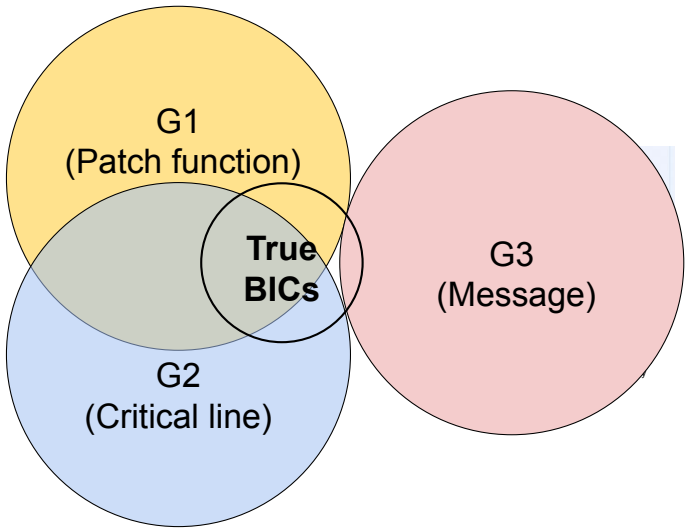

Figure 4.4: Candidate Generators

The quality of the candidate lists can significantly impact the accuracy of the final BIC identification. On the one hand, too many commits in the list will simultaneously increase the likelihood of errors and the cost. On the other hand, missing relevant commits

leads to false negatives. As a result, we would ideally like the list to (1) contain the true BIC, and (2) be small enough. In practice, these two goals are hard to achieve simultaneously. We will present our key design below to strike a good balance.

Function-based Generator. As used in the baseline method (§4.3.1), the most commonly used generator in existing work is based on patched function(s) in the bug-fix commit, where all historical commits modifying the same function(s) are selected as candidates. This strategy is effective as bug-inducing and -fix commits frequently modify the same function(s).

Critical-line-based Generator. First, it recognizes lines that are truly relevant to the vulnerability logic (*i.e.,* critical lines). To achieve this, we utilize LLM's ability to comprehend both code and natural language to recognize critical lines, which are far more accurate than heuristic-based approaches used in existing work. We also provide LLM with the full definitions of the patched functions, as a part of prompt engineering, to better facilitate its understanding of vulnerability logic. Second, we will only treat historical commits that modify critical lines as BIC candidates.

Conceptually, we would like an LLM to focus on particular parts of the code that pertain to the vulnerability, whether they are part of the patched functions or changes to a global variable definition (if it is included in the code diff). It turns out that it is a non-trivial task. As mentioned, prior work often relies on overly simplistic heuristics to define critical lines. For example, all deleted lines within a function are considered critical[58], or every line in the patched functions is defined as critical[194]. We would like to generalize it and improve it, with the help of LLMs.

In particular, we divide patches into three types and apply tailored strategies using LLMs to identify critical lines:

(1) Patches with deleted lines. Deleted lines in a bug-fix commit are often related to the vulnerability, so in this case, we narrow our scope of critical line identification to the deleted lines (excluding trivial ones like comments) to improve efficiency. However, if LLMs recognize no critical lines among those deleted, we expand our scope to the whole patched function.

(2) Patches with only added lines. If a patch has only added lines, previous solutions, such as VSZZ simply give up. However, we would extract critical lines from the entire modified function/struct. Specifically, we would feed the whole patch, including the code diff and commit message, as well as the complete definitions of the affected functions. For example, if the patch merely adds a range check for a variable (such as an array index), the LLM can analyze the commit message and the function's surrounding code to identify critical statements related to this variable (e.g., an array access with the index, where the out-of-bounds (OOB) error occurs). These critical statements are often modified in the BIC.

(3) Patches with only reordered lines. These patches merely change the line positions (*e.g.,* adjust the critical section length by moving the lock/unlock statements). Here vulnerabilities are usually caused by improper relative positioning of two lines, one being the line modified by the patch and the other whose relative order to the modified line has changed. Therefore, merely focusing on the presence of modified lines is insufficient to determine whether a vulnerability exists. The introduction of a vulnerability is often

closely related to the other line. Therefore, for such patches, we extract critical lines from the modified lines and the affected context statements (the statements whose relative position to the modified line has been altered after applying the patch). For example, if a patch moves a Lock() call to an earlier position in the function, thereby extending the scope of the lock to include more statements, the statements newly encompassed by the lock after the patch are considered affected context statements. These often include critical statement related to the vulnerability.

Commit-message-based Generator. As discussed in §4.3.1, neither of the above generators can correctly include the BIC candidate if it has no overlap (regarding the modified code) with the bug-fix commit. To address this issue, we design the third generator based on commit messages, from which we extract valuable information regarding the vulnerability. More specifically, we try to extract the following information from the commit message:

(1) Function/struct/variable names. They could indicate the actual location of the vulnerable code or global variables.

(2) Commit hashes. Some commit messages directly reference earlier (BIC) commits by their hashes. We also include names of modified functions by the bug-fix as keywords, though technically they are not extracted from the bug-fix commit messages. They are useful because even the BIC may not modify the same functions, it might still modify their callers which contain their names in the code. To avoid redundant execution, we disregard all functions or structs modified by the Bug-fix commit (which have already been tracked by the first two generators). After running the three above candidate generators independently,

we obtain *three* candidate lists at the end of this stage, which will be fed as input to the next stage (§4.3.4).

4.3.4 BIC Filtering

In §4.3.3, we generate three lists of candidates, at this stage, we try to pick *one* most likely BIC from *each* list, resulting in *up to three* final BIC candidates (it is posible that no BIC is selected from a certain list) for the next stage (§4.3.5). One straightforward way to pick the BIC from a candidate list, as mentioned in §4.3.1, is to inspect each commit in reverse chronological order and stop when one is recognized as the BIC. However, this leads to a high positive rate because the "most likely" BIC infers that it can only be reliably identified from a comparison of multiple potential ones. Therefore, we design our BIC filtering process to be composed of two sub-phases: the pre- and post-filtering.

(1) Pre-Filtering. For *every* commit in a candidate list, we prompt it with the original bug-fix commit to LLM for a decision regarding whether it *could* be the BIC. This will result in multiple potential BICs selected by the LLM.

(2) Post-Filtering. The LLM is then instructed to perform a *comparative assessment* of all selected BICs in the pre-filtering phase, to finally pick *one* most likely BIC per candidate list.

This design gives LLMs sufficient opportunities to review all candidate commits and carefully compare them for better-informed decisions, significantly boosting the BIC identification accuracy, compared to baseline early stop solution.

4.3.5 Result Finalization

The last BIC filtering stage (§4.3.4) outputs up to three potential BICs selected from multiple candidate lists, while we still need to finalize our decision by picking one final BIC. To achieve this, our procedure is similar to the last stage (§4.3.4). Specifically, we present all BIC candidates (up to three) to the LLM for a comparative evaluation, in order to reach a final BIC decision as SymBisect's output. Note that though rarely the case, it is possible that SymBisect eventually fails to output any BIC (*e.g.,* zero candidates were selected in the previous BIC filtering or this result finalization process).

As mentioned in §4.3.1, we have three methods of generating BIC candidates. They can complement each other. As a result, SymBisect adopts all three of the aforementioned BIC candidate generators. Our design is to have them work independently initially. Later on, we will attempt to pick the final result with Result Finalization. Note that we do not want to merge all the candidates into the same set initially and then have LLM pick one. This is because such a set will be too large which will hurt the accuracy. As mentioned in §4.3.1, the number of candidates produced by the function-based generator is already large, limiting the LLM's accuracy in picking the right BIC. It is therefore beneficial to keep the set of candidates produced by critical-line-based and commit-message-based methods separate. This way, if the correct BIC is located in either of the two sets, it will likely be correctly identified by the LLM. Again, the function-base generator can be viewed as a backup option. In case the correct BIC is present in only its result, then at least we would still have a chance to identify it.

4.4 Implementation

We implement a prototype of SymBisect with 5,331 LoC in Python. In this section, we discuss some noteworthy implementation details.

Function-Based Candidate Generation. One can use a git command to track all commits that modify a specific function: `git log <commit_hash> -L:<funcname>:<filename>`. However, it can miss some commits when the function has been renamed or the file that contains the function has been renamed. To address this limitation, we developed a script to track all commits that modify the given file/function more comprehensively, correctly handling the renaming issues. For each commit, we then extract the functions modified by it, enabling us to obtain the complete list of commits modifying the specific function.

Patch Type Classification. We implement a Python script to first determine the type of patch (e.g., those with only added lines). This is relatively straightforward. We first extract and ignore all changes relating to comments, and then can easily classify patches into those with only added lines and those with deleted lines (we do not differentiate the patches with only deleted lines). Among the patches with deleted lines, if there are also added lines, we then use a simple string-match-based heuristic to identify reordered statements. Specifically, we consider a patch as reordering changes if and only if all the changes are related to reordering. In other words, all the removed lines must show up as added lines in another location verbatim.

After collecting this information, we first obtain historical commits that modify the same files as the bug-fix commit, then for each commit, we check whether it matches any

of the extracted information with the commit message (*e.g.,* has the same hash, change/call the mentioned function, *etc.*). If so, we also consider it as a BIC candidate, which can be missed by function-based or critical-line-based generators.

Majority Voting. As mentioned in §4.3.2, we employ the majority voting mechanism to battle the well-known self-consistency issue of LLMs (*i.e.,* multiple runs with the same input produce contradictory results). Specifically, we execute the LLM session for a set number of times (defaulted at 7) to reach a consensus.

LLM Models. In our implementation, we primarily use OpenAI o1 (`o1-preview-2024-09-12`) as the main LLM. We also evaluate other models, including GPT-4o (`gpt-4o-2024-08-06`) and the open-source, Llama 3 (`nvidia/llama-3.1-nemotron-70b-instruct`). The results of these evaluations are presented in Section 4.5.3. The specific prompts used in each step are included in the appendix.

4.5 Evaluation

In this section, we evaluate SymBisect to answer the following research questions:

- RQ1: How accurately does SymBisect identify BICs?
- RQ2: How does SymBisect compare against other state-of-the-art BIC identification methods?
- RQ3: How does each design point of SymBisect contribute to its final performance?

Dataset. We evaluate SymBisect against Linux kernel CVEs. Several key considerations inform this choice: (1) Linux kernel is one of the most important and widely used software,

its ecosystem contains numerous downstream distributions potentially impacted by N-day vulnerabilities, highlighting the importance of an accurate bug bisection, (2) The kernel also has one of the most complex codebases, containing a wide range of vulnerabilities reported daily by security practitioners. We believe the diversity and complexity of Linux kernel CVEs can rigorously test SymBisect's accuracy and reliability. Note that despite our choice, SymBisect by design is agnostic to the target software or vulnerability types.

Given the sheer number of Linux kernel CVEs and the high cost of advanced LLM tokens (e.g., o-1 preview), we randomly sampled 100 CVEs in each of 2023 and 2024 (200 in total). We specifically include CVEs in 2024 as they are published after the LLM knowledge cut-off date, validating whether SymBisect's result is influenced by the LLM's pre-existing knowledge about the CVEs (The results show that there is no significant difference. SymBisect demonstrated similar accuracy on CVEs from 2023 and 2024). We also included CVEs in 2023 because the CVE assignment criteria became more relaxed starting from 2024 (*e.g.,* many non-security issues also had CVEs assigned) [15, 27, 11]. Testing these CVEs demonstrates SymBisect's accuracy on security vulnerabilities more reliably.

Ground Truth. To get the ground truth (*i.e.,* the correct BIC for a specific vulnerability), we intentionally include in our dataset only those CVEs whose fix commit has a *fixes tag* [12], which points to the BIC(s) given by kernel developers. We then manually verify them according to our BIC definition and assemble the ground truth. For example, in the patch 3f77c7d605b2[1], the developer incorrectly treated the first occurrence of the bug function as the BIC; even though the bug originated from its subsequent update. After manual

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/ patch/?id=3f77c7d605b2

verification, we identified and corrected 12 cases with inaccurate fix tags. It is important to note that fix tags are merely used to provide the ground truth that is otherwise difficult to obtain - we remove the fixes tags from bug-fix commits before testing SymBisect.

Comparison Targets. We extensively compare SymBisect with three state-of-the-art tools covering different bug bisection methodologies:

(1) PoC-based bisection. SymBisect[210] is a state-of-the-art PoC-based bisection tool. It generates various guidance from PoC execution traces and uses principally guided under-constrained symbolic execution to confirm the bug's existence. However, it only supports limited vulnerability types and relies on PoCs — unavailable for most Linux kernel CVEs. Even then, we would like to see how SymBisect compares to SymBisect using its evaluation dataset, which SymBisect supports very well. This is an interesting experiment that can showcase the performance differences between the symbolic reasoning (in SymBisect) and LLM's reasoning (in SymBisect).

(2) Patch-based bisection with SZZ-style algorithms. As mentioned in §4.2, SZZ-style algorithms generally rely on the assumption that BIC will initialize lines deleted in the bug-fix commits. We select VSZZ [58] — the state-of-the-art open-source tool in this domain — as a comparison target, we configure it with default options specified in its tutorial [30].

(3) Patch-based bisection with vulnerable code clone detection. These methods are based on code similarity comparison between the vulnerable pre-fix version and a specific target version to probe the first vulnerable version (§4.2). V0Finder [194] is a latest tool in this

Dataset	Tools	Correct	Incorrect	Accuracy
SymBisect (200 CVEs)	SymBisect	182	18	91%
	V0Finder	66	134	33%
	VSZZ	102	98	51%
SymBisect (32 syzbot bugs)	SymBisect	29	3	90.6%
	SymBisect	24	8	75%

Table 4.1: **The results of BIC identification**

Tools	TP	FP	FN	Precision	Recall	F-1 Score
SymBisect	4121	151	146	96.5%	96.6%	96.5%
V0Finder	1594	56	2748	96.6%	36.7%	53.2%
VSZZ	4140	1660	85	71.4%	98.0%	82.6%

Table 4.2: **The results of vulnerable versions detection**

direction, we configure it with its default options [28] in our comparison.

4.5.1 Accuracy of SymBisect (RQ1)

Accuracy of BIC Identification. Table 4.1 shows the results of BIC identification with different tools, SymBisect consistently achieves the highest accuracy of more than 90%, outperforming other state-of-the-art tools by significant margins (*i.e.,* 25.6% - 58%). Specifically, SymBisect accurately identified the correct BICs for the two motivating examples mentioned in Section 4.2. Note that the comparison with SymBisect is based on

Phase	Reason	Num
Candidate commit Generation	BIC changed different files	8
	Insufficient info in commit messages	2
BIC Filtering	Not Pick groundtruth as final BIC	4
Result Finalization	Not Pick groundtruth as final BIC	4

Table 4.3: **The reasons of SymBisect's inaccuracy**

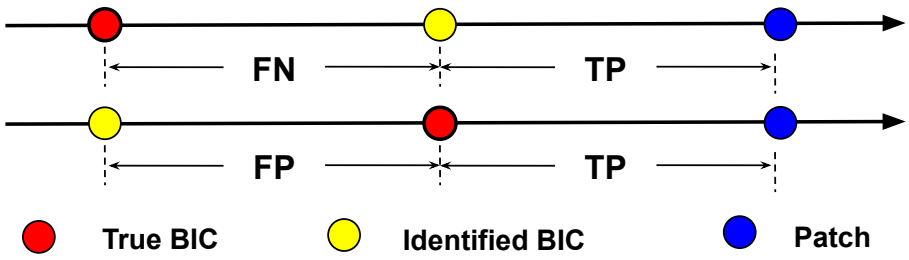

Figure 4.5: Explanation of TP/FP/FN

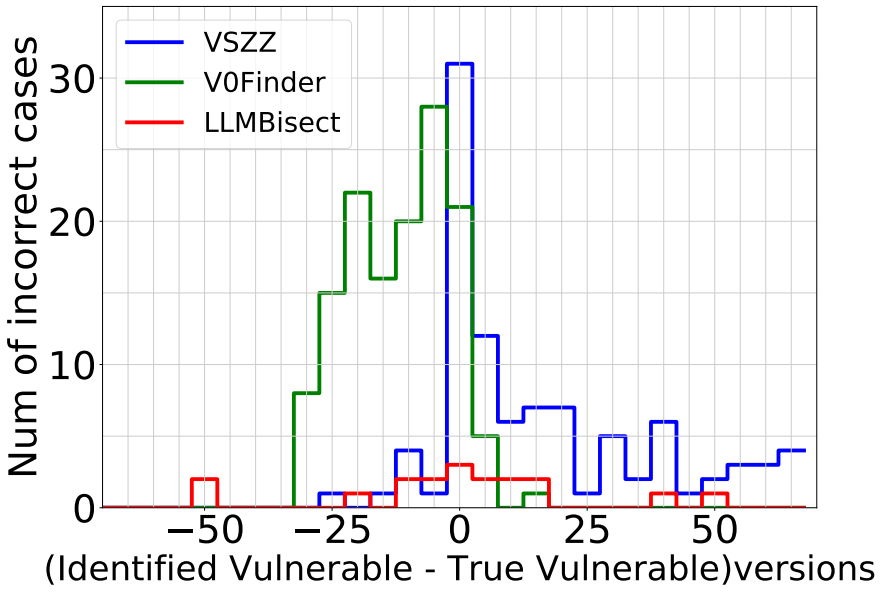

Figure 4.6: Distribution of inaccurate cases over version distances

SymBisect's own dataset due to its reliance on PoCs and specific vulnerability types, as mentioned previously in §4.5. These results show SymBisect's superior accuracy, even on dataset originally designed for other tools. We will describe the comparison results in detail in §4.5.2.

Accuracy of Vulnerable Version Detection. One common application of bug bisection is to determine the software versions affected by a vulnerability, informing downstream

developers for timely patch porting [211]. From this perspective, solely evaluating the accuracy of BIC identification has its limitations. For example, if a vulnerability is fixed in version 6.0 but introduced in version 5.0, a tool that identifies the introduction of the bug in version 5.1 or 5.19 would both be considered inaccurate from the perspective of BIC identification accuracy. However, the impact of such inaccuracies on downstream users can vary significantly.

Therefore, in addition to verifying whether our tool accurately identifies bug-inducing commits, we also evaluate the accuracy of identifying vulnerable versions. Specifically, once the BIC is determined, we can identify all vulnerable versions on the Linux mainline branch, i.e., versions between the BIC and the patch, considering only major releases such as v5.0, v5.1. By comparing the vulnerable versions derived from the true BIC with those derived from the BIC identified by our tool, we calculate the tool's false positives (FP), false negatives (FN), and true positives (TP) for this task. As Figure 4.5 shows, once we identify the BIC, we can determine the numbers of TP, FN, and FP based on its relative position to the true BIC and the patch commit. However, the number of TNs depends on the manually selected starting point (e.g., whether we start counting from v2.6 or v4.0) and is not a fixed value. Therefore, TNs are not included in our statistics.

Figure 4.6 shows the distribution of inaccurate cases for different methods in terms of FP/FN versions. The X-axis represents the number of FP or FN vulnerable versions for each case (e.g., 10 indicates a case where the method produced 10 false positive vulnerable versions, and -5 indicates a case where the method produced 5 false negative vulnerable

versions). Note that, as shown in Figure 5, a single method cannot produce both FP and FN for the same case.

We group the inaccurate cases into intervals of 5 based on their FP/FN counts and plot the number of cases in each group on the Y-axis. From the figure, we can observe that VSZZ produces a large number of false positive versions, V0Finder generates many false negative versions, whereas SymBisect significantly reduces both false positives and false negatives.

N-day vulnerabilities. In our evaluation dataset, which includes 200 randomly selected Linux CVEs, we identified 22 N-day bugs that were not patched in the latest Linux LTS patches (a total of 45 bug-LTS pairs). We confirmed our findings with Linux maintainers, which validated the effectiveness of our results. Most of them remain unpatched because the maintainers lack the time and resources to address conflict issues. A smaller portion has been overlooked by maintainers for various reasons. We believe that if our method is applied to cases without a 'Fixes' tag, more unpatched N-day vulnerabilities would be discovered. We leave this exploration as future work.

Inaccuracy Analysis. As shown in Table 4.1, SymBisect has 18 inaccurate bisection cases out of the 200 CVEs, after inspecting each, we summarize 4 underlying reasons arising in 3 different phases of SymBisect (Fig. 4.3), as listed in Table 4.3. We now detail these reasons by phase.

Phase I: Candidate Generation. SymBisect will miss the correct BIC (*i.e.,* false negative) if it is not included in the initial candidate list, 10 failure cases belong to this category. Specifically, for 8 of them, the BIC and bug-fix commits modify completely

different files, making it difficult to recognize the correct BIC candidates without incurring a high cost (*e.g.,* we need to enumerate virtually *all* commits for all files in the codebase.). In the remaining 2 cases, the BIC and bug-fix commit modify different functions, structs, or variables within the same file, however, our candidate generator fails to correlate them based on the bug-fix commit message, which does not contain enough hints (*e.g.,* the vulnerable function name) to locate the remotely related BIC.

Phase II: BIC Filtering. In this phase, LLMs first try to identify (multiple) potential BICs from a specific generator's candidate list, then select *one* BIC from multiple by comparing them. We have 4 failure cases where the true BIC does not survive this filtering process. Upon further investigation, we found that the failure is mainly because of the excessive number of potential BICs to filter (*e.g.,* 84.25 on average for these 4 cases vs. 36.5 for all). This confirms our design consideration (§4.3.1) that more candidate commits can decrease the accuracy, besides increasing the costs. We also observed that LLM's self-consistency issue contributes to 3 of these failure cases, where the correct BIC can be selected in some LLM runs but not in others.

It is worth noting that we do not have any inaccurate cases in the pre-filtering phase (*e.g.,* LLMs fail to pick the correct BIC from the generator's candidate list at beginning), this confirms our observation (§4.3.1) that LLM is less likely to make FNs when deciding whether an individual commit is a potential BIC.

Phase III: Result Finalization. Phase II selects one BIC from each of three generators' candidate lists, resulting in three final BIC candidates. Then, the result finalizer further selects one BIC from these three. 4 failure cases are due to that the correct BIC does not

Reason	Inaccurate Cases	Solved in SymBisect
BIC changed different functions	19	9
Only focus on deleted lines	28	27
Not identified critical lines	6	5
Flawed Heuristic	45	40
Total	98	81

Table 4.4: **The reasons of VSZZ method failed**

survive this final "1/3" selection process. We observed that the failure here is again related to LLMs' self-consistency (*e.g.,* correct BICs can survive in *some* runs).

4.5.2 Comparison against SOTA Tools (RQ2)

As shown in Table 4.1 and Figure 4.6, SymBisect significantly outperforms other state-of-the-art tools regarding accuracy. In this section, we provide an in-depth analysis of these tools' inaccuracies and how SymBisect improves over them.

VSZZ. VSZZ identifies the BIC as the earliest commit that initializes the lines deleted by the bug-fix commit. If the bug-fix does not delete any lines, the commit initializing the file modified by the bug-fix will be treated as the BIC. We group VSZZ's inaccurate cases based on flaws in this heuristic algorithm and discuss how SymBisect addresses them.

Flaw 1. VSZZ fundamentally assumes that deleted lines in the bug-fix commit are related to the vulnerability's root cause, so the BIC must introduce these lines. However, the BIC can actually be within completely different functions (*e.g., 19* such cases in our dataset) or irrelevant to those deleted lines (*6* such cases in our dataset).

Flaw 2. The bug-fix commit can have no deleted lines, in this case, the heuristic of "treating

Reason	Inaccurate Cases	Solved in SymBisect
BIC changed different functions	19	9
Not identified critical lines	84	80
Flawed Heuristic	31	28
Total	134	117

Table 4.5: **The reasons of V0Finder method failed**

Reason	Inaccurate Cases	Solved in SymBisect
Under-constraint Symbolization	5	4
Scalability	3	3
Total	8	7

Table 4.6: **The reasons of SymBisect method failed**

the line-initialization commit as BIC" is oversimplified and highly inaccurate. 28 of bug-fix commits in our dataset have no deleted lines.

Flaw 3. The BIC may *modify* but *not initialize* the deleted lines in the bug-fix commit, violating VSZZ's heuristic. We observed *45* such cases in our dataset.

The above flaws stem from VSZZ's reliance on hardcoded, simplified, and code-oriented heuristics. SymBisect, on the other hand, utilizes LLM's deep and flexible understanding of vulnerability logic (*e.g.,* recognize critical lines) to identify BICs, with minimal assumptions, *e.g.,* the presence (*Flaw 2*) and significance (*Flaw 1*) of deleted lines and BIC's operation (*Flaw 3.*). Furthermore, SymBisect takes advantage of full patch context, including the commit messages, to extract valuable information for BIC locating, significantly addressing *Flaw 1.* As a result, SymBisect resolves 81 out of 98 VSZZ's inaccurate cases.

V0Finder. V0Finder treats the pre-patched version of functions modified in the bug-

fix commit as vulnerable, it then compares it to all previous versions syntactically, by essentially a *whole-function* strict string match with certain abstraction and normalization. All identical historical versions will also be treated as vulnerable, while the BIC is the commit turning a non-vulnerable version into vulnerable. We detail V0Finder's weaknesses as follows.

Flaw 1. Similar to *Flaw 2* of VSZZ, the BIC may not make changes to the same functions as in the bug-fix commit (*e.g.,* 19 such cases in our dataset), rendering V0Finder's patch-function-based BIC probing invalid.

Flaw 2. V0Finder's syntactical similarity calculation is unaware of semantics and vulnerability logic. Consequently, it will likely identify a historical commit as the BIC wrongly, as long as it makes *any* changes (that cannot be normalized or abstracted away by V0Finder's string matching algorithm) in the function patched by the bug fix, These changes may not relate to the vulnerability at all (*e.g.,* not on the critical lines of the vulnerability) — 84 of V0Finder's inaccurate cases are due to this, or relate to but not introduce the vulnerability — 31 failure cases are due to this.

As mentioned before, SymBisect addresses these shortcomings by making decisions based on the understanding of the vulnerability logic with the help of LLMs and its comprehensive consideration of the patch contexts. As a result, SymBisect resolves 117 out of 134 V0Finder's inaccurate cases.

SymBisect. SymBisect decides whether a specific vulnerability affects a software version with under-constrained symbolic execution, guided by hints extracted from PoC execution traces for better scalability. Despite its reliance on PoC and limited support for vulnerability

types, we identify issues impacting its accuracy on its own evaluation dataset (that we use for our comparison).

Flaw 1. Under-constraint symbolic execution assumes overly relaxed constraints (and often infeasible) of program variables unknown in its analysis scope, *e.g.,* global variables initialized outside of the local analyzed function(s). This results in over-approximation of program behaviors, for instance, a software version can wrongly be recognized as vulnerable. SymBisect fails in 5 cases in our dataset due to this reason.

Flaw 2. Symbolic execution is known to be expensive. To address the scalability issue, SymBisect utilizes information (*e.g.,* promising paths) extracted from PoC execution traces to guide its symbolic execution. However, this guide may be incomplete or inaccurate, leading to missed vulnerable paths and/or conditions, eventually causing inaccuracies in BIC identification. We observed 3 such inaccurate cases in the SymBisect evaluation dataset.

SymBisect, unlike SymBisect, does not rely on the expensive symbolic execution for BIC identification. Instead, its decision is based on LLM's profound understanding of the vulnerability logic, from both code changes and commit messages, avoiding the above difficulties.

4.5.3 Ablation Study (RQ3)

Effectiveness of Design Points.

As discussed in §4.3.1, our final design results from multiple iterations and refinements of a baseline workflow. During this process, we adopt different effective design points that *all* improve SymBisect's accuracy. To demonstrate it, we start with the

baseline method and gradually integrate each of our design points, observing the change in BIC detection accuracy. We show the results in Fig. 4.7, as can be seen, the accuracy steadily improves as more design points are adopted (*e.g.,* from 30.5% to 91%). In the remainder of this section, we detail the reasons behind these improvements by analyzing each intermediate configuration in Fig. 4.7.

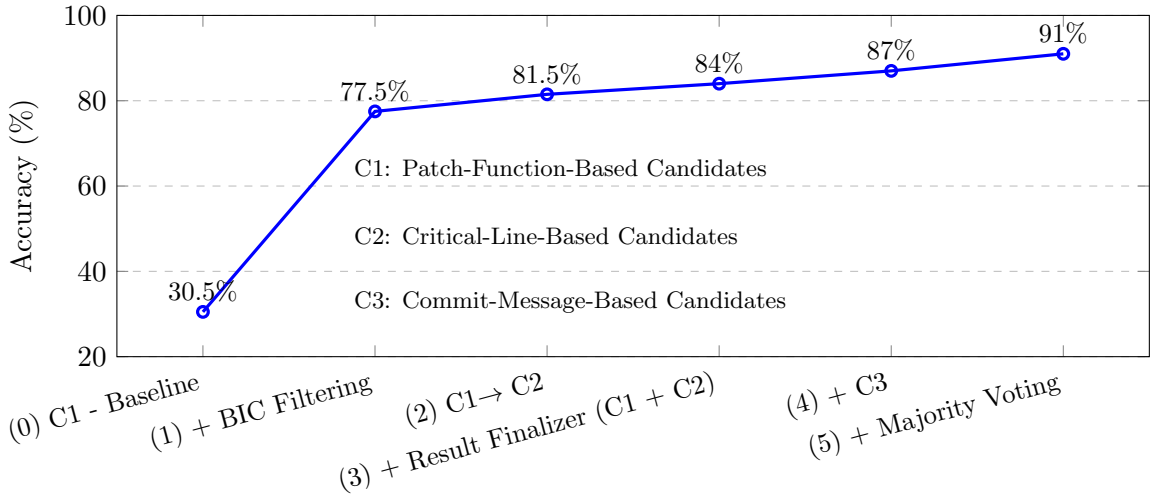

Figure 4.7: Ablation Study with Different Design Points.

(0) The Baseline Method. As described in §4.3.1, the most straightforward baseline method inspired by existing work is to let LLM inspect each commit (reverse chronologically) that touches the same function(s), i.e., candidates are generated using the patch-function-based generator alone. The first identified BIC will be output as the final result. As analyzed in §4.3.1, this approach has a low accuracy (30.5% in Fig. 4.7) mainly due to LLM's high false positive rate in single-commit BIC decision and missing true BIC with single generator.

(1) Added: BIC Filtering. We then adopt the BIC comparative filtering process (§4.3.4), where all potential BICs are identified and then compared by the LLM to determine the

most likely one. As shown in Fig. 4.7, this significantly improve the accuracy compared to the strawman workflow (30.5% → 77.5%).

(2) Replaced: C1 → C2. Patch-function-based candidate generation (*i.e., C1* in Fig. 4.7) can result in too many candidates, confusing the LLM and eventually reducing accuracy. We show that a more fine-grained critical-line-based strategy (*C2* in Fig. 4.7 to replace *C1*, detailed in §4.3.3) increases the accuracy from 77.5% to 81.5%.

(3) Added: Result Finalizer. As discussed in §4.3.1, critical-line-based candidate generation (*C2* in Fig. 4.7) is more precise, however, it can also miss true BICs if some critical lines are missed. Our solution is to combine *C2* and *C1* with the *result finalizer* (§4.3.5), this design further improves the accuracy (81.5% → 84% in Fig. 4.7).

(4) Added: Commit-Message-Based Candidate Generation. Neither *C1* nor *C2* captures BICs having no code overlaps with the corresponding bug-fix commits, as mentioned in §4.3.1. We thus develop another strategy that seeks implicitly connected BICs from the commit messages of the bug-fix (*C3* in Fig. 4.7, detailed in §4.3.3). As shown in Fig. 4.7, this improves the accuracy to 87% from the previous configuration.

(5) Added: Majority Voting. As mentioned before (§4.5.1), the well-known self-consistency issue of LLMs can negatively impact our accuracy, when the correct decision is not yielded in the first run. To address this, we incorporate the *majority voting* mechanism which selects the most frequent answer among multiple LLM runs in the result finalizer. This further improves SymBisect's accuracy compared to the previous configuration (*i.e.,* 87% → 91% in Fig. 4.7).

Model	Inaccurate Cases	Accuracy
OpenAI o1	18	91%
GPT-4o	69	65.5%
LLama3.2	58	71%

Table 4.7: **The accuracy with different LLM models**

Patch Information	Inaccurate Cases	Accuracy
Commit Message+ Code Change	18	91%
Code Change	58	71%

Table 4.8: **The accuracy with/without commit message**

The Role of Commit Messages.

One of SymBisect's major advantages is its utilization of the full patch information, including both code changes and natural language commit messages. Besides *C3* in Fig. 4.7 for candidate initialization, commit messages also help LLMs make more informed decisions when inspecting each commit for BIC identification. To quantitatively understand the commit message's impact, we strip the commit messages of all commits and re-run our evaluation. Note that the impact is multi-front: (1) the commit-message-based generator basically no longer works, (2) the critical-line-based generator is substantially weaker because the LLM can no longer benefit from the commit messages to understand the logic of the bug, and (3) the selection of the BICs is also weaker because the LLM can no longer benefit from the description of the purpose of the candidate commits. As shown in Table 4.8, the gap in accuracy is significant: 71% vs 91%.

Different LLMs.

SymBisect's design is agnostic to the underlying LLM, nonetheless, we conduct a comparative evaluation by swapping between three widely used LLMs: OpenAI o1, GPT-4o, and LLama 3, covering both commercial and open-source models. The evaluation results are summarized in Table 4.7. As can be seen, OpenAI o1 achieves the highest accuracy (91%) likely due to its enhanced reasoning capability, followed by LLama 3 (71%) and GPT-4o (65.5%). We found that the majority of inaccuracies occur during the process of comparing multiple suspected BICs and selecting the final result (specifically, during the Post-Filtering and Result Finalizer stages). For example, GPT-4o produced a total of 49 inaccurate cases across these two steps. This suggests a gap for different models in tasks which requires extensive reasoning on multiple code snippets and commit message.

4.6 Related Work

The Application of LLMs in Program Analysis. Recent research has explored the integration of LLMs into static analysis to enhance its effectiveness in code comprehension and bug findings [181, 179, 129, 180]. LLMs have also been employed to understand and generate code comments, documentation, and system logs, improving code readability and maintainability [113, 130, 95]. The integration of LLMs in program analysis represents a significant advancement in software engineering, offering tools that enhance productivity, code quality, and security.

PoC-based vulnerable version identification. SymBisect [210] leverages under-constrained symbolic execution to determine whether a specific software version

contains a given vulnerability, enabling the identification of BICs. However, SymBisect supports only specific types of functions and requires an existing PoC. Dai et al.[75] proposed a PoC migration approach that takes an initial PoC as input and adapts it to identify other affected versions; however, it is specifically designed for user-space programs.

SZZ Methods. SZZ (short for Śliwerski, Zimmermann, and Zeller) [170] is an algorithm designed to identify bug-inducing commits in version control systems, also called B-SZZ. It identifies earlier changes at the location of a bug fix as bug-inducing commits. However, its straightforward approach struggles to handle complex bugs effectively. To address this limitation, AG-SZZ [124] incorporates an annotation graph to exclude non-semantic changes, such as whitespace, comments, and formatting adjustments, thereby reducing false positives. MA-SZZ [74] further improves on this by filtering out meta-changes like branch modifications and file attribute updates, ensuring that only source code changes are analyzed. V-SZZ [58] expands the algorithm's scope by targeting vulnerabilities introduced in earlier software versions. NEURAL-SZZ [175] leverages a Heterogeneous Graph Attention Network (HAN) to capture semantic relationships between lines of code, enhancing precision in tracing bug origins. However, it is limited to Java and exhibits a relatively high false positive rate. Combining advanced techniques like NEURAL-SZZ and V-SZZ can significantly improve bug-tracing accuracy, while AG-SZZ and MA-SZZ remain practical solutions for simpler scenarios.

Vulnerable code clone detection. Vulnerable code clone detection is a specialized type of code clone detection [49, 160, 161, 168, 87]. It involves identifying pieces of source code

in software systems that are similar to or identical to code fragments known to have security vulnerabilities. They usually perform similarity comparisons on what they define as vulnerability-related code (usually a few lines within the patch function or the entire function) [123, 109, 64, 200, 212, 64, 194]. However, vulnerability-related code extraction based on simple heuristics may not effectively extract the code most relevant to vulnerabilities. Similarly, similarity comparisons based on predefined rules cannot always accurately determine whether a vulnerability truly exists in a given version. Essentially, these methods do not compare vulnerabilities based on their logical structure. Our evaluation demonstrates that, for this type of analysis, the state-of-the-art methods have limited accuracy in complex programs (such as Linux).

4.7 Conclusion

In conclusion, we introduced SymBisect, a novel, LLM-driven bug bisection pipeline that effectively pinpoints bug-inducing commits in Linux kernel. By combining both code changes and commit-message insights, SymBisect overcomes the limitations of traditional patch-based methods, which often fail to capture the true scope and context of a vulnerability. Our results underscore the potential of large language models to streamline vulnerability detection, reducing the window in which attacks can occur.

Chapter 5

Conclusions

In this work, we set out to address the challenges posed by N-day vulnerabilities in open-source software, with a special focus on the Android kernel ecosystem. Our research makes three major contributions:

1. Comprehensive Measurement Study: We conducted a large-scale measurement study of patch propagation across multiple layers—from the Linux mainline to various downstream kernels used in Android devices. Our analysis revealed that delays in patch deployment are common, sometimes taking months or even years to fully reach end users. These delays increase the window during which attackers can exploit known vulnerabilities. We analyzed the root causes behind the significant delays, including flaws in the overall Android patch propagation mechanism. We shared our findings with major players in the Android ecosystem, such as Google, Qualcomm, and Samsung. We were invited to present at the Google Android Security Organization and collaborated with a team to deploy our tool, which helped spur Google to propose solutions to the identified issues.

2. Novel Bug Bisection Technique (SymBisect): Traditional bug bisection methods often struggle because they rely solely on dynamic testing or simple heuristics. To overcome these limitations, we introduced SymBisect—a novel tool that uses under-constrained symbolic execution to trace vulnerabilities back to their bug-inducing commits. By isolating the exact change that introduced a vulnerability, SymBisect can significantly reduce false positives and improve the accuracy of bisection.

3. Integration of Large Language Models (LLMBisect): Recognizing that commit messages and code context contain valuable information, we enhanced our approach with LLM-driven techniques. These models help analyze both code diffs and natural language descriptions, allowing our system to better capture the true context of a vulnerability. This integration further improves the precision of bug-inducing commit identification and speeds up the patch porting process .

Together, these contributions offer a more complete solution to the twin problems of delayed patch propagation and inaccurate bug bisection. By demonstrating that a combination of precise symbolic analysis and modern LLM capabilities can effectively narrow the exposure window for N-day vulnerabilities, our work lays a strong foundation for both improved automated security analysis and practical vulnerability remediation.

While our methods show promising results, challenges remain. For instance, although LLMBisection is applicable to various bugs and achieves high accuracy, it is designed only to identify the upstream-affected versions. It does not consider the impact of downstream customization on vulnerability exposure. Even if the affected versions are known, customization may prevent the direct application of upstream patches,

necessitating manual work by maintainers. Therefore, to fully address the N-day vulnerability problem, our approach should be integrated with other techniques, such as automatic patch translation.

Ultimately, this dissertation provides actionable insights and practical tools that can help the open-source community reduce the risk of N-day vulnerabilities. By accelerating the detection and patching of vulnerabilities, our approach contributes to building a more secure and resilient software ecosystem.

Bibliography

[1] https://git-scm.com/.

[2] https://subversion.apache.org/.

[3] https://android.googlesource.com/.

[4] https://www.kernel.org/.

[5] https://cve.mitre.org/.

[6] Android Operating System. https://www.android.com/.

[7] Android Security Bulletin. https://source.android.com/security/bulletin/.

[8] Android Security Bulletin—January 2020. https://source.android.com/security/bulletin/2020-01-01.

[9] BinDiff. https://www.zynamics.com/bindiff.html.

[10] CVE: Vulnerabilities By Year. https://www.cvedetails.com/browse-by-date.php.

[11] Data for May 2024 CVE. https://github.com/jgamblin/monthlyCVEStats/blob/main/2024/May/May2024.ipynb.

[12] Fixes Tag. https://docs.kernel.org/process/submitting-patches.html.

[13] Github Annual Report. https://octoverse.github.com/.

[14] Google wants Android to use regular Linux kernel, potentially improving updates and security. https://www.androidpolice.com/2019/11/19/google-wants-android-to-use-regular-linux-kernel-potentially-improving-updates-and-

[15] kernel.org Added as CVE Numbering Authority (CNA). https://www.cve.org/Media/News/item/news/2024/02/13/kernel-org-Added-as-CNA.

[16] KLEE bug type. https://mailman.ic.ac.uk/pipermail/klee-dev/2020-April/001983.html.

[17] Linux Kernel Faces Reduction in Long-Term Support Due to Maintenance Challenges. https://www.linuxjournal.com/content/linux-kernel-reduction-longterm-support.

[18] Linux stable kernel patch rules. https://www.kernel.org/doc/Documentation/process/stable-kernel-rules.rst.

[19] National Vulnerability Database. https://nvd.nist.gov/.

[20] NetworkX Python Package. https://networkx.github.io/.

[21] Pixel Update Bulletins. https://source.android.com/security/bulletin/pixel.

[22] Qualcomm customer-specific releases for Android. https://wiki.codeaurora.org/xwiki/bin/QAEP/release.

[23] Security Patch for CVE-2015-8955. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=8fff105e13041e49b82f92eef034f363a6b1c071.

[24] SymBisect Source Code. https://github.com/zhangzhenghsy/SymBisect.

[25] Syzbot Bisection. https://android.googlesource.com/platform/external/syzkaller/+/HEAD/docs/syzbot.md#bisection.

[26] Syzbot Bisection Motivation. https://lore.kernel.org/all/CACT4Y+Y3nN=nLEkHXLFcX7vxp_vs1JrD=8auJ3cX9we6TQHO+w@mail.gmail.com/T/#u.

[27] The kernel becomes its own CNA. https://lwn.net/Articles/961961/.

[28] V0Finder Source Code. https://github.com/WOOSEUNGHOON/V0Finderpublic.

[29] Valgrind. http://valgrind.org/.

[30] VSZZ Source Code. https://figshare.com/ndownloader/files/31748777.

[31] VUDDY Source Code. https://github.com/squizz617/vuddy.

[32] Vulnerability Definition. https://csrc.nist.gov/glossary/term/software_vulnerability.

[33] What to do about CVE numbers. https://lwn.net/Articles/801157/.

[34] Android Security Bulletin—February 2017. https://source.android.com/security/bulletin/2017-02-01, 2019.

[35] HiSilicon. http://www.hisilicon.com/, 2019.

[36] Huawei-firmware. http://huawei-firmware.com/phone-list/, 2019.

[37] Latest Official Android ROMs. https://www.cnroms.com/, 2019.

[38] MediaTek still has no plans to release source code to the community. https://www.xda-developers.com/mediatek-source-code-release-no-plans/, 2019.

[39] MIUI Global ROM. http://c.mi.com/oc/miuidownload/index, 2019.

[40] Oppo Software Updates. https://oppo.custhelp.com/app/soft_update, 2019.

[41] Sammobile. www.sammobile.com, 2019.

[42] Samsung Exynos. https://www.samsung.com/semiconductor/minisite/exynos/products/all-processors/, 2019.

[43] Samsung to lay off nearly 300 as it closes Austin unit project. https://www.statesman.com/business/20191101/samsung-to-lay-off-nearly-300-as-it-closes-austin-unit-project, 2019.

[44] Sony Software binaries. https://developer.sony.com/develop/open-devices/downloads/software-binaries, 2019.

[45] Stock ROM files. https://stockromfiles.com/, 2019.

[46] Rui Abreu, Franjo Ivančić, Filip Nikšić, Hadi Ravanbakhsh, and Ramesh Viswanathan. Reducing time-to-fix for fuzzer bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1130. IEEE, 2021.

[47] Adam Conway. How Monthly Android Security Patch Updates Work. https://www.xda-developers.com/how-android-security-patch-updates-work/.

[48] Yehuda Afek, Omer Ben-Shalom, and Anat Bremler-Barr. On the structure and application of BGP policy Atoms. 2002.

[49] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[50] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How long do vulnerabilities live in the code? a {Large-Scale} empirical measurement study on {FOSS} vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, 2022.

[51] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. http://tools.ietf.org/html/rfc2581, 1999.

[52] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. Fonte: Finding bug inducing commits from failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 589–601. IEEE, 2023.

[53] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Self-Tuning Wireless Network Power Management. *Wireless Networks*, 2005.

[54] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4255–4269, 2021.

[55] Cornelius Ascherm, Sergej Schumilo, Tim Blazytko, Robert Gawlik, , and Thorsten Holz. Fuzzing with input-to-state correspondence. NDSS, 2019.

[56] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. ICSE'14.

[57] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, October 1997.

[58] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. V-szz: automatic identification of version ranges affected by cve vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2352–2364, 2022.

[59] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. ICSM'98.

[60] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. *ACM SIGPLAN Notices*, 49(10):491–506, 2014.

[61] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. Orca: Differential bug localization in {Large-Scale} services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509, 2018.

[62] Swapnil Bhartiya. Greg Kroah-Hartman Explains How the Kernel Community Is Securing Linux. https://www.linux.com/topic/linux/greg-kroah-hartman-explains-how-kernel-community-securing-linux-0/.

[63] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.

[64] Benjamin Bowman and H Howie Huang. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.

[65] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forcasting and Control*. Prentice Hall, 3rd edition, 1994.

[66] Hong Chao, Chun-Yu Ho, Tin Cheuk Leung, and Travis Ng. To Root or Not to Root? The Economics of Jailbreak. Technical report, Social Science Research Network (SSRN), 2013.

[67] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. USENIX Security, 2020.

[68] Code Aurora. Android for MSM Project. https://wiki.codeaurora.org/xwiki/bin/QAEP/.

[69] Code Aurora. Android releases. https://wiki.codeaurora.org/xwiki/bin/QAEP/release.

[70] Code Aurora. Security Bulletin. https://www.codeaurora.org/category/security-bulletin/page/3.

[71] International Data Corporation. Smart Phone OS Market Share. http://www.idc.com/promo/smartphone-market-share/os.

[72] Dawson Engler Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008) December 8-10, 2008, San Diego, CA, USA.

[73] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. USENIX Security'12.

[74] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Trans. Softw. Eng.*, 2017.

[75] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3300–3317, 2021.

[76] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. ASE'07.

[77] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

[78] Dawson Engler David A Ramos. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security, 2015.

[79] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[80] Dan Nguyen-Huu Dharmesh Thakker, Max Schireson. Tracking the explosive growth of open-source software. https://techcrunch.com/2017/04/07/tracking-the-explosive-growth-of-open-source-software/.

[81] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[82] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *USENIX Security Symposium*, pages 869–885, 2019.

[83] Niall Douglas. User Mode Memory Page Allocation: A Silver Bullet For Memory Allocation? Technical report, 2011.

[84] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[85] Clément Elbaz, Louis Rilling, and Christine Morin. Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[86] eng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[87] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 516–527, 2020.

[88] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. 2019.

[89] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. ASIACCS'17.

[90] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. CCS '16.

[91] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138. ACM, 2006.

[92] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[93] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, 2008.

[94] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[95] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, New York, NY, USA, February 2024. Association for Computing Machinery.

[96] Google. Distribution dashboard. https://developer.android.com/about/dashboards.

[97] Google. Google syzbot. https://syzkaller.appspot.com/upstream/.

[98] Google. Google syzkaller. https://github.com/google/syzkaller.

[99] Google. Learn when you'll get Android updates on Pixel phones and Nexus devices. https://support.google.com/pixelphone/answer/4457705?hl=en.

[100] Google. Stable Kernel Releases & Updates - Security. https://source.android.com/devices/architecture/kernel/releases#security.

[101] Google Project Zero. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[102] Google Project Zero. Issue 1942: Android: Use-After-Free in Binder driver. https://bugs.chromium.org/p/project-zero/issues/detail?id=1942.

[103] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE Computer Society, 2023.

[104] Yongzhong He, Yiming Wang, Sencun Zhu, Wei Wang, Yunjia Zhang, Qiang Li, and Aimin Yu. Automatically identifying cve affected versions with patches and developer logs. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[105] Thong Hoang, Julia Lawall, Richard J Oentaryo, Yuan Tian, and David Lo. Patchnet: a tool for deep patch classification. ICSE'19 Demonstrations.

[106] He Huang, Amr M. Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. ASIACCS'17.

[107] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 618–635. IEEE, 2016.

[108] Frank K Hwang, Dana S Richards, and Pawel Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[109] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[110] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[111] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[112] Yuning Jiang, Manfred Jeusfeld, and Jianguo Ding. Evaluating the data inconsistency of open-source vulnerability repositories. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.

[113] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proceedings of the ACM on Software Engineering*, 1(FSE):137–160, July 2024.

[114] Jonathan Corbet. Bringing the Android kernel back to the mainline. https://lwn.net/Articles/771974/.

[115] Hyuckmin Kwon Jonghoon Kwon Heejo Lee Hongzhe Li. A scalable approach for vulnerability discovery based on security patches. in applications and techniques in information security. In Applications and Techniques in Information Security (ATIS 2014). Springer, Berlin, Heidelberg, 109–122.

[116] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[117] Wooseok Kang, Byoungho Son, and Kihong Heo. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1695–1708, 2022.

[118] PCWorld Katherine Noyes. Open Source Software Is Now a Norm in Businesses. https://www.pcworld.com/article/228136/open_source_software_now_a_norm_in_businesses.html.

[119] Android Kernel. How Android common kernels developed. https://source.android.com/devices/architecture/kernel/android-common, 2019.

[120] Linux Kernel. How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html, 2019.

[121] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[122] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011.

[123] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.

[124] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006.

[125] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. ICSE'07.

[126] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag.

[127] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[128] Frank Li and Vern Paxson. A large-scale empirical study of security patches. CCS'17.

[129] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages (PACMPL), Volume 8, Issue OOPSLA1*, 8(OOPSLA1), 2024.

[130] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. Only diff Is Not Enough: Generating Commit Messages Leveraging Reasoning and Action of Large Language Model. *Proceedings of the ACM on Software Engineering*, 1(FSE):745–766, July 2024.

[131] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, pages 310–320. IEEE Press, 2012.

[132] Xingyu Li, Zheng Zhang, Zhiyun Qian, Trent Jaeger, and Chengyu Song. An investigation of patch porting practices of the linux kernel ecosystem. *arXiv preprint arXiv:2402.05212*, 2024.

[133] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[134] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. ACSAC'16.

[135] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection.

[136] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.

[137] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[138] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[139] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, 2012.

[140] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130, 2000.

[141] Seyed Mohammadjavad, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Q. Charm: Facilitating dynamic analysis of device drivers of mobile systems. USENIX Security, 2018.

[142] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 188–197. IEEE, 2021.

[143] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*, pages 692–708. IEEE, 2015.

[144] Michal Nazarewicz. A Deep Dive into CMA. https://lwn.net/Articles/486301/.

[145] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[146] OpenSignal. Android Fragmentation Visualized. http://opensignal.com/reports/2015/08/android-fragmentation/, 2015.

[147] James O'Toole. Mobile apps overtake PC Internet usage in U.S. http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/.

[148] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, pages 93–104, 2006.

[149] Idrees Patel. Xiaomi Still Hasn't Released Kernel Sources for the Mi A1. https://www.xda-developers.com/xiaomi-not-released-kernel-sources-mi-a1/, 2018.

[150] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. Tfuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*. IEEE, 2018.

[151] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[152] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. Oakland'15.

[153] J. Pewny, F. Schuster, C. Rossow, L. Bernhard, and T. Holz. Leveraging semantic signatures for bug search in binary programs. ACSAC'14.

[154] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456. ACM, 2010.

[155] piunikaweb. Asus releases botched up kernel sources for Zenfone Max M2 family on launch day. https://piunikaweb.com/2018/12/12/asus-releases-botched-up-kernel-sources-for-zenfone-max-m2-family-on-launch-day/, 2018.

[156] Qualcomm. Security Bulletin. https://www.qualcomm.com/company/product-security/bulletins.

[157] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security'15.

[158] David A. Ramos and Dawson R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 49–64. USENIX Association, 2015.

[159] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M Germán, and Jesus M Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25:1294–1340, 2020.

[160] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[161] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168, 2016.

[162] Samsung. Knox Deep Dive: Real-time Kernel Protection (RKP). https://www.samsungknox.com/en/blog/knox-deep-dive-real-time-kernel-protection-rkp, 2019.

[163] Eunjin Choi Heejo Lee Seunghoon Woo, Hyunji Hong. Movery: A precise approach for modified vulnerable code clone discovery from modified open-source software components. USENIX Security, 2022.

[164] Seulbae Kim Heejo Lee Seunghoon Woo, Sunghan Park and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021.

[165] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.

[166] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE international conference on machine learning and applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[167] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. Precise (un) affected version analysis for web vulnerabilities. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[168] G Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655, 2021.

[169] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Oakland'16.

[170] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[171] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[172] SRLabs. The Android patch ecosystem – Still fragmented, but improving. https://srlabs.de/bites/android-patch-gap-2020/.

[173] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 63–78, Austin TX USA, November 2024. ACM.

[174] T-mobile. Software updates: LGV30. https://www.t-mobile.com/support/devices/android/lg-v30v30-plus/software-updates-lg-v30v30-plus.

[175] Lingxiao Tang, Lingfeng Bao, Xin Xia, and Zhongdong Huang. Neural szz algorithm. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1024–1035, 2023.

[176] Yuan Tian, Julia Lawall, and David Lo. Identifying Linux bug fixing patches. ICSE'12.

[177] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. ICSE'12.

[178] David Trabish, Shachar Itzhaky, and Noam Rinetzky. A bounded symbolic-size model for symbolic execution. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE*, pages 1190–1201. ACM, 2021.

[179] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. CORE: Resolving Code Quality Issues using LLMs. *Proceedings of the ACM on Software Engineering*, 1(FSE):789–811, July 2024.

[180] Chengpeng Wang, Yifei Gao, Wuqi Zhang, Xuwei Liu, Qingkai Shi, and Xiangyu Zhang. LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis, December 2024. arXiv:2412.14399 [cs].

[181] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. Sanitizing Large Language Models in Bug Detection with Data-Flow. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3790–3805, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[182] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. USENIX Security, 2021.

[183] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[184] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. Graphspd: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 604–621. IEEE Computer Society, 2022.

[185] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–124, 2017.

[186] Emil Wåreus and Martin Hell. Automated cpe labeling of cve summaries with machine learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 3–22. Springer, 2020.

[187] Huihui Wei and Ming Li. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *IJCAI*, pages 2840–2846, 2018.

[188] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 262–273, 2016.

[189] Yang Wen, Jicheng Cao, and Shengyu Cheng. Ptracer: A linux kernel patch trace bot.

[190] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

[191] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. Hiddencpg: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the ACM Web Conference 2022*, pages 755–766, 2022.

[192] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. {MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3037–3053, 2022.

[193] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. MOVERY: A precise approach for modified vulnerable code clone discovery from modified open-source software components. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3037–3053. USENIX Association, 2022.

[194] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0finder: Discovering the correct origin of publicly reported software vulnerabilities. In *USENIX Security Symposium*, pages 3041–3058, 2021.

[195] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. NDSS, 2020.

[196] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 23:2866–2900, 2018.

[197] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[198] Julia Wunder, Alan Corona, Andreas Hammer, and Zinaida Benenson. On nvd users' attitudes, experiences, hopes, and hurdles. *Digital Threats: Research and Practice*, 5(3):1–19, 2024.

[199] Sen Chen Feng Wu Tianming Liu Xiapu Luo Xian Zhan, Lingling Fan and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In Proceedings of the 43rd International Conference on Software Engineering (ICSE), 2021.

[200] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[201] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. Large Language Models for Cyber Security: A Systematic Literature Review, July 2024. arXiv:2405.04760 [cs].

[202] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[203] Tuba Yavuz. Sift: A tool for property directed symbolic execution of multithreaded software. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433–443, 2022.

[204] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul L. Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *ESEC/FSE*, pages 221–232. ACM, 2020.

[205] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul L. Yu. Progressive scrutiny: Incremental detection of UBI bugs in the linux kernel. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[206] Haibo Zhang and Kouichi Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.

[207] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.

[208] Xiaodong Zhang, Zijiang Yang, Qinghua Zheng, Yu Hao, Pei Liu, and Ting Liu. Tell you a definite answer: Whether your data is tainted during thread scheduling. *IEEE Trans. Software Eng.*, 46(9):916–931, 2020.

[209] Xiaodong Zhang, Zijiang Yang, Qinghua Zheng, Pei Liu, Jialiang Chang, Yu Hao, and Ting Liu. Automated testing of definition-use data flow for multithreaded programs. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 172–183. IEEE Computer Society, 2017.

[210] Zheng Zhang, Yu Hao, Weiteng Chen, Xiaochen Zou, Xingyu Li, Haonan Li, Yizhuo Zhai, and Billy Lau. {SymBisect}: Accurate bisection for {Fuzzer-Exposed} vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2493–2510, 2024.

[211] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3649–3666, 2021.

[212] Deqing Zou, Hanchao Qi, Zhen Li, Song Wu, Hai Jin, Guozhong Sun, Sujuan Wang, and Yuyi Zhong. Scvd: A new semantics-based approach for cloned vulnerable code detection. In *DIMVA*, pages 325–344. Springer, 2017.

[213] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. {SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, 2022.