Automated Code-Behavior and -Semantic Understanding for Security

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Qiushi Wu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Advisor: Kangjie Lu

September, 2023

Acknowledgements

The Ph.D. journey has been the most enriching and introspective chapter of my life. Not only has it expanded the boundaries of my intellectual curiosity, but it has also profoundly shaped my understanding of dedication, resilience, and the pursuit of knowledge. Each page of this thesis carries the fruits of my research alongside the imprints of personal growth, challenges, collaboration, and the realization of a dream nurtured over five years. Throughout my Ph.D. journey, numerous individuals have offered unwavering support; I extend my heartfelt thankfulness to all of them.

First and foremost, I extend my heartfelt appreciation to my advisor, Prof. Kangjie Lu. Your expertise and insights have been indispensable throughout my research journey. Your guidance not only honed my research skills but also played a pivotal role in sculpting the trajectory of my academic career.

Secondly, I'd like to sincerely thank my committee members: Prof. Anand Tripathi, Prof. Stephen McCamant, and Prof. John Sartori. Our interactions during the preliminary exam and thesis proposal enlightened my learning journey. Your constructive feedback and probing questions have been instrumental in molding and steering the direction of my research.

Thirdly, the success and breadth of my research projects owe much to collaborating with talented researchers from various groups. I wish to extend my gratitude to the following collaborators: From the University of Minnesota: Dr. Navid Emamdoost, Dr. Wenjia Zhao, Weiheng Bai, Qingyang Zhou, Kefu Wu, Bowen Wang, Aditya Pakki, Le Peng, and Yang He. From Indiana University Bloomington: Prof. Xiaojing Liao and Yue Xiao. From UC Riverside: Prof. Zhiyun Qian, Prof. Srikanth Krishnamurthy, Dr. Shitong Zhu, Xingyu Li, and Zheng Zhang. From Zhejiang University: Prof. Shouling Ji and Dinghao Liu. From Nanjing University: Prof. Bing Mao and Jianhao Xu. From

Georgia Institute of Technology: Feng Xiao. From IBM Thomas J. Watson Research Center: Dr. Hani Jamjoom and Dr. Zhongshu Gu. Our discussions and brainstorming were not only thought-provoking but were also the bedrock of our fruitful collaborations. I cherish and value each of these partnerships immensely. Additionally, I'd like to thank all other co-authors and collaborators who have contributed to various publications during my Ph.D. journey. Your insights and collaborations have enriched the scope and depth of my work.

Lastly, my sincere appreciation goes to my family and friends for their unwavering support. To my family, Yi Wen, Linqing Wu, Meng Wu, and Jingjing Wu: your firm belief in me, coupled with your steadfast love, has served as my compass throughout this journey. To my dear friends, notably Bowen Wang, Shi Chen, Jingfan Guo, Yuan Yao, Zhixuan Yu, Tianci Song, Yao Gong, and Jiacheng Liu: your emotional support and the refreshing breaks from the demands of research have been valuable. Even though our academic paths didn't align, your emotional backing has been critical to my Ph.D. journey. Additionally, I cannot forget to mention my emotional support dog, Xiaoweiba, who provided tireless companionship and comfort during the most challenging times.

Dedication

To all who bravely chase their dreams.

Abstract

There has been a growing focus on strengthening program security to protect software ecosystems, especially in light of the swift expansion of available programs in the software supply chain. Static program analysis, embraced by both the industry and academia, allows for an in-depth examination of a program without executing it, making it pivotal in enhancing software security.

Static program-analysis techniques delve deeply into various aspects of programs, whether at the source code, binary, or intermediate representation (IR) level. They can dissect data dependencies, control flow, type information, memory operations, cache activities, function calls, and more, which disclose the low-level semantics of a program. By harnessing this information, one can pinpoint security vulnerabilities, examine patches, or simulate the execution behavior of a program. The capabilities of static program analysis are rooted in the foundational principles of programming language and compiler theories.

However, traditional static analysis also has shortcomings, particularly in grasping the high-level semantics of programs. For example, it struggles to extract complex programming logic rules, such as the privilege prerequisites for accessing specific variables or functions. Furthermore, when faced with a function, such as fread(), the static analysis cannot accurately interpret its high-level behavior—reading a file. However, understanding such high-level code behaviors is pivotal for in-depth analysis of the security facets of programs. For example, distinguishing between confidential and non-confidential data is crucial since each demands distinct privilege protection mechanisms. Recognizing such a difference necessitates a sophisticated grasp of the program's high-level semantics. Consequently, bridging the gap between high-level code behaviors and low-level code semantics is imperative for bolstering the security of real-world programs. And over the past few years, we have done the following work to bridge this gap.

Firstly, we utilized general behavioral rules of code, summarized with statistical methods, to minimize the reliance on high-level code semantics. Specifically, we introduced HERO, a system designed to detect *Disordered Error Handling* (DiEH) bugs. It operates on a fundamental programming principle: error cleanup functions should be invoked in a

stack-like order. Leveraging this rule, HERO could pinpoint numerous error-handling related bugs, such as use-after-free, without tapping into the high-level semantics of programs.

Our second work used security rules and formal definitions to analyze code behaviors. Specifically, we introduced SID to evaluate the security impacts bugs based on their corresponding patches. The driving concept behind SID is that both the impact of a patch and violations of security rules, such as out-of-bound access, can be framed as constraints solvable through automated methods. Consequently, SID can accurately distinguish between patches related to security and those unrelated to it. In this project, the high-level semantics of the code are extracted by human interpretation and later evaluated using formal methods.

Besides these, we also leveraged machine learning (ML) models to decipher the behaviors of functions semi-automatically. Specifically, we developed DiffCVSS to discern the correlation between functions and CVSS metrics by analyzing both function descriptions and vulnerability narratives. On the other hand, we employed GNNIC to probe the similarity among functions by scrutinizing their call graphs, function names, and utilized types, all with the assistance of graph neural networks. In these two projects, the high-level semantics of the code are summarized and analyzed using natural language processing techniques combined with machine learning methodologies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Problem Statement

Nowadays, the utilization of program static-analysis techniques to enforce program review is a commonly adopted practice within software development. This is because the program static-analysis allows developers to effectively analyze the low-level code semantics without runtime execution, which can significantly enhance the code review process. Traditional program static analysis techniques typically analyze the code semantics at a low level and formally focus on the behavior of instructions, values, basic blocks, pointers, etc. However, such techniques may not be able to fully understand the intent or purpose of the code. For example, it cannot understand how different parts of the code interact with each other to accomplish a specific task or how the code is intended to be used by end-users.

Unfortunately, securing programs necessitates the utilization of rules based on such high-level code behavior. Such rules include, but are not limited to, the proper performance of permission checks prior to critical usage, validation of input from untrusted areas, and the proper release of resources post-usage. Thus, to effectively conduct a program static analysis and secure software, it is necessary to bridge the gap between code behaviors and low-level code semantics.

The primary objective of this thesis is to explore and reconcile the disparity between code behaviors and low-level code semantics and further use them to analyze software security issues. The code behaviors of modern software programs remain highly complex,

rendering a complete understanding of all the programs elusive even for experienced researchers. Thus, such complexity precludes the ability to rely entirely on machine learning models to comprehensively understand the code behaviors and further guide static program analysis.

In light of the complexity of code behaviors in modern programs, we structure our research into three primary steps. The first step is minimizing the reliance on overly complex code behaviors to simplify the analysis process. The second step involves a manual examination of the code, during which we will summarize essential code behaviors and extract insights. The final step leverages machine learning and natural language processing techniques to assist the analysis of code behaviors, supplementing traditional program analysis methods. Through such approaches, we hope to gain a deeper understanding of the code behaviors in modern software and further develop more robust and efficient analysis approaches.

1.2 Thesis Overview

Table 1.1: A summary of techniques for bridging the gap between high-level code behaviors and low-level code semantics.

Technique	**Code Behavior Examples**
Reduce leveraging the code behaviors	Pairly used functions[4]
Specifications-based approach	Security checks[5, 6], Bond check[7], Initialization[7] Failure-handling (FH) [4, 5, 6]
ML and NLP based approach	Privileges required functions [8] User-interaction related functions [8] Adjacent-network related functions [8] Abstract behavior of functions [9]

As demonstrated in Table 1.1, the research endeavors to achieve the research goal through three distinct solutions: (1) use general rules or statistical methods for reduction in the reliance on code behaviors, (2) a specifications-based approach for the analysis

of code behaviors, and (3) the utilization of ML and NLP techniques to uncover code behaviors semi-automatically. The following subsections will provide a comprehensive overview of these techniques.

1.2.1 Reduce Leveraging the Code Behaviors

One prevalent approach to circumvent the difficulties associated with high-level code behaviors during program analysis is the utilization of statistical analysis or general rules. For example, Crix [5], a static-analysis tool developed by us that can detect missing security check bugs through cross-checking, which employs the collection of a substantial number of similar code snippets and the identification of deviating cases as potential bugs. Such techniques are relatively easy to be applied for complex code bases. However, due to relying on a significant number of similar cases for the statistical analysis to be effective, cross-checking may not be suitable for cases that are rarely encountered. In contrast, HERO [4] represents an alternative approach, which is based on a general design rule, namely the adherence to a stack-like order in the invocation of function pairs, to circumvent the need for complex code behaviors or large-scale statistical analysis. Unlike the cross-checking technique previously discussed, this approach can identify bugs in functions that are not frequently used.

HERO: Mitigating Reliance on code behaviors through the General Rules for Function Pairing

Software programs may frequently encounter various errors such as allocation failures. Error handling aims to gracefully deal with the errors to avoid security and reliability issues, thus it is prevalent and vital. However, because of its complexity and corner cases, error handling itself is often erroneous, and prior research has primarily focused on finding bugs in the *handling* part, such as incorrect error-code returning or missing error propagation.

In this work, we propose and investigate a class of bugs in error-handling code from a different perspective. In particular, we find that programs often perform "cleanup" operations before the actual error handling, such as freeing memory or decreasing refcount. Critical bugs occur when these operations are performed (1) in an incorrect order, (2) redundantly, or (3) inadequately. We refer to such bugs as *Disordered Error Handling*

(DiEH). Our investigation reveals that DiEH bugs are not only common but can also cause security problems such as privilege escalation, memory corruption, and denial-of-service. Based on the findings from the investigation, we then develop a system, HERO (Handling ERrors Orderly), to automatically detect DiEH. The core of HERO is a novel technique that precisely pairs both common and custom functions based on the unique error-handling structures, which allows us to infer expected cleanup functions. With HERO, we found 239 DiEH bugs in the Linux kernel, the FreeBSD kernel, and OpenSSL, which can cause security and reliability issues. The evaluation results show that DiEH is critical and widely exists in system software, and HERO is effective in detecting DiEH. We also believe that the precise function pairing is of independent interest in other research areas such as temporal-rule inference and race detection.

1.2.2 Specification-based Approach to Find Code Behaviors

Nevertheless, defining and utilizing the code behaviors are unavoidable in many instances. In such cases, we present several specification-based methodologies for defining code behaviors by examining low-level code semantics. Specifically, such an approach involves collecting basic information from program analysis, including instructions, values, basic blocks, slices, and functions, which is subsequently used to define code behaviors through the combination of experiences, published documentation, and pre-defined code behaviors. For example, in addition to utilizing cross-checking, Crix also incorporates manual definitions for several code behaviors, including failure handling and security checks. Similarly, SID [7] also incorporates the definition of code behaviors pertinent to the most commonly encountered types of vulnerabilities, such as variable initialization. Based on such functionalities semantics, SID can precisely differentiate security bugs from non-security bugs.

SID: Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison

A bug is a vulnerability if it has security impacts when triggered. Determining the security impacts of a bug is important to both defenders and attackers. Maintainers of large software systems are bombarded with numerous bug reports and proposed patches, with missing or unreliable information about their impact. Determining which few bugs

are vulnerabilities is difficult, and bugs that a maintainer believes do not have security impact will be de-prioritized or even ignored. On the other hand, a public report of a bug with a security impact is a powerful first step towards exploitation. Adversaries may exploit such bugs to launch devastating attacks if defenders do not fix them promptly. Common practice is for maintainers to assess the security impacts of bugs manually, but the scaling and reliability challenges of manual analysis lead to missed vulnerabilities.

We propose an automated approach, SID, to determine the security impacts for a bug given its patch, so that maintainers can effectively prioritize applying the patch to the affected programs. The insight behind SID is that both the effect of a patch (either submitted or applied) and security-rule violations (e.g., out-of-bound access) can be modeled as constraints that can be automatically solved. SID incorporates rule comparison, using under-constrained symbolic execution of a patch to determine the security impacts of an un-applied patch. SID can further automatically classify vulnerabilities based on their security impacts. We have implemented SID and applied it to bug patches of the Linux kernel and matching CVE-assigned vulnerabilities to evaluate its precision and recall. We optimized SID to reduce false positives, and our evaluation shows that, from 54K recent valid commit patches, SID detected 227 security bugs with at least 243 security impacts at a 97% precision rate. Critically, 197 of them were not reported as vulnerabilities before, leading to delayed or ignored patching in derivative programs. Even worse, 21 of them are still unpatched in the latest Android kernel. Once exploited, they can cause critical security impacts on Android devices. The evaluation results confirm that SID's approach is effective and precise in automatically determining security impacts for a massive stream of bug patches.

1.2.3 ML and NLP-based Approach to Find CVE-related Functions

Machine learning (ML) and natural language processing (NLP) techniques become necessary when the code behaviors of code are challenging to encapsulate through low-level semantics. An example of such an approach is DiffCVSS [8], a tool that employs NLP techniques to analyze the commit messages from the program Git history and the vulnerabilities' descriptions. Thus allowing for the identification and classification of functions related to specific vulnerabilities' exploitability metrics (e.g., identification of functions related to user interaction). By examining the vulnerability call-chains in

various operating systems (OSes), DiffCVSS can compare the severity of a vulnerability on these systems.

DiffCVSS: OS-Aware Vulnerability Prioritization via Differential Severity Analysis

The Linux kernel is quickly evolving and extensively customized. This results in thousands of versions and derivatives. Unfortunately, the Linux kernel is quite vulnerable. Each year, thousands of bugs are reported, and hundreds of them are security-related bugs. Given the limited resources, the kernel maintainers have to prioritize patching the more severe vulnerabilities. In practice, Common Vulnerability Scoring System (CVSS) [10] has become the standard for characterizing vulnerability severity. However, a fundamental problem exists when CVSS meets Linux—it is used in a "one for all" manner. The severity of a Linux vulnerability is assessed for only the mainstream Linux, and all affected versions and derivatives will simply honor and reuse the CVSS score. Such an undistinguished CVSS usage results in underestimation or overestimation of severity, which further results in delayed and ignored patching or wastes of the precious resources. In this paper, we propose OS-aware vulnerability prioritization (namely DiffCVSS), which employs differential severity analysis for vulnerabilities. Specifically, given a severity-assessed vulnerability, as well as the mainstream version and a target version of Linux, DiffCVSS employs multiple new techniques based on static program analysis and natural language processing to differentially identify whether the vulnerability manifests a higher or lower severity in the target version. A unique strength of this approach is that it transforms the challenging and laborious CVSS calculation into automatable differential analysis. We implement DiffCVSS and apply it to the mainstream Linux and downstream Android systems. The evaluation and user-study results show that DiffCVSS is able to precisely perform the differential severity analysis, and offers a precise and effective way to identify vulnerabilities that deserve a severity reevaluation.

1.2.4 Using GNN and NLP to Summarize the Abstract Behavior of Functions

Analyzing and quantifying the behavior of functions is crucial for understanding the high-level semantics of code. Recognizing the behavior of functions enables us to assess their semantic similarity, detect similar coding issues, and more. In our work, we harness the abstract similarity of functions to filter indirect-call targets. The underlying concept is that the target functions of an indirect-call site should possess comparable functionality. For instance, if a function pointer is "p->read(...)", then all its target functions should be related to reading operations. Building on this insight, we developed GNNIC [9] to identify functions that exhibit abstract similarity. This similarity is then used to filter indirect call targets and identify related coding issues, among other applications.

GNNIC: Finding Long-Lost Sibling Functions with Abstract Similarity. Generating accurate call graphs for large programs, particularly at the operating system (OS) level, poses a well-known challenge. This difficulty stems from the widespread use of indirect calls within large programs, wherein the computation of call targets is deferred until runtime to achieve program polymorphism. Consequently, compilers are unable to statically determine indirect call edges. Recent advancements have attempted to use type analysis to globally match indirect call targets in programs. However, these approaches still suffer from low precision when handling large target programs or generic types.

This paper presents GNNIC, a Graph Neural Network (GNN) based Indirect Call analyzer. GNNIC employs a technique called *abstract-similarity search* to accurately identify indirect call targets in large programs. The approach is based on the observation that although indirect call targets exhibit intricate polymorphic behaviors, they share common abstract characteristics, such as function descriptions, data types, and invoked function calls. We consolidate such information into a representative abstraction graph (RAG) and employ GNNs to learn function embeddings. Abstract-similarity search relies on at least one anchor target to bootstrap. Therefore, we also propose a new program analysis technique to locally identify valid targets of each indirect call. Starting from anchor targets, GNNIC can expand the search scope to find more targets of indirect calls in the whole program. The implementation of GNNIC utilizes LLVM and GNN, and we evaluated it on multiple OS kernels. The results demonstrate that GNNIC

outperforms state-of-the-art type-based techniques by reducing 86% to 93% of false target functions. Moreover, the abstract similarity and precise call graphs generated by GNNIC can enhance security applications by discovering new bugs, alleviating path-explosion issues, and improving the efficiency of static program analysis. The combination of static analysis and GNNIC resulted in finding 97 new bugs in Linux and FreeBSD kernels.

1.3 Our Contributions

In this subsection, we present the code behaviors discovered through our current research and the corresponding security-related achievements achieved via the implementation of such code behaviors

- **Proposing DiEH bugs.** While prior research primarily focused on the "handling" part, we find that, in the error paths, programs often perform "cleanup" operations before actually handling the errors. For the example shown in Figure 2.1, when the function video_register_device() (line 13) encounters an error, the code releases the pointer vfd (line 24) and unregisters the device (line 26) before passing the error to its caller. As the cleanup operations, these functions must be called correctly; otherwise, the program is vulnerable to bugs such as use-after-free. Buggy cases include calling cleanup functions (1) in an incorrect order, (2) redundantly, and (3) inadequately. We refer to such bugs as *Disordered Error Handling* (DiEH). While prior research studied inadequate error handling such as missing memory release [11] and missing refcount release [12], redundant and incorrect-order error-handling problems are unexplored.

- **An effective detection system—HERO.** Based on our empirical study of DiEH bugs, we identify three challenges in their detection. First, DiEH represents the root causes of a wide class of semantic bugs in error-handling code from a different perspective, so the detecting rules are undefined yet. Second, a DiEH case may not be harmful, so we need to distinguish and remove harmless cases. Third, by nature, code paths containing DiEH bugs often involve path conditions (e.g., return-value checks), so path-feasibility testing is required to ensure that the paths are valid. To address these problems, we model DiEH and propose HERO (Handling ERrors

Orderly). HERO is equipped with multiple techniques such as *scalable symbolic summaries* for eliminating infeasible paths and *dependency reasoning* for removing harmless incorrect-order DiEH cases. HERO also provides rankings to facilitate the final manual confirmation for DiEH bugs.

- **A study of security bugs and patches.** The boundary between bugs and vulnerabilities can be unclear. We first study the differences between bugs and vulnerabilities. We then model patches for common security bugs, including missing/wrong bound check, missing pointer nullification, missing initialization, and missing permission check. The modeling enables us to define the security impacts of bugs and thus confirm security bugs.

- **Symbolic rule comparison for determining security impacts.** We propose SID to determine the security impacts of bugs automatically. The core of SID is *symbolic rule comparison* which employs differential and under-constrained symbolic execution to precisely confirm the security impacts that a patch fixes. In addition, SID also provides details about the security impacts to facilitate bug fixing.

- **Mapping functions to CVSS metrics.** We train a set of classifiers to map functions to the CVSS exploitability metrics based on their descriptions in Linux kernel and further leverage transfer learning to transform the semantic knowledge learned from Linux to the Android domain.

- **Identifying and matching vulnerability paths for CVEs.** Based on CVE information, DiffCVSS employs static program analysis and NLP to precisely identify the corresponding vulnerability paths (from an entry point to the vulnerable function) and match them between Linux and Android. We believe that identifying vulnerability paths is a useful technique that can enable further research such as patch generation and testing, and impact analysis.

- **Analyzing abstract similarity of functions.** We propose *abstract similarity* analysis as a general approach to refine indirect-call targets or assist other program analysis techniques. It performs well in cases (e.g., general types, large programs, unscalable pointer analysis) where existing approaches suffer from. It is complementary to existing approaches, such as type or pointer-based analysis, and can be

used together with them.

- **Developing graph-based techniques for indirect call identification.** We propose *representative abstraction graph* (RAG), which is designed to capture diverse information (e.g., textual description, nested function calls, and data types) of functions and can be directly processed by GNN. Such an integrated and GNN-compatible representation may serve as a generic technique to enable many applications of GNN in program analysis. We also propose *scoped unique-name matching* as a precise technique to identify anchor functions for indirect calls.

- **Evaluating on a spectrum of security applications.** We present a comprehensive evaluation of the security applications with GNNIC. Our evaluation first showcases the effectiveness of GNNIC in improving the precision of bug identification. As a demonstration, we found 97 new NULL-pointer dereference bugs in well-established OS kernels, including the Linux kernel and FreeBSD kernel, by incorporating abstract similarity of functions with traditional static analysis techniques. In addition, we present multiple other security applications that can significantly benefit from the precise results generated by GNNIC. Overall, our comprehensive evaluation demonstrates the potential of GNNIC in improving the security of software systems.

The rest sections of this thesis delve into each of these works in detail.

Chapter 2

HERO: Understanding and Detecting Disordered Error Handling with Precise Function Pairing

A program may encounter various errors at runtime, including hardware errors (e.g., disk corruption), software errors (e.g., an unlock without a lock), and invalid inputs. To avoid crashes and insecure operations, the error-handling mechanisms capture and gracefully deal with errors. As such, error handling plays a key role in ensuring the security and reliability of programs. Also, error-handling code is very prevalent; for example, according to our study, there are more than 400K occurrences of error handling in about 18K source files in the Linux kernel.

Unfortunately, error-handling code itself is often erroneous. In particular, EIO [13] even shows that error-handling code is "occasionally" correct. After checking the latest 100 CVE-assigned vulnerabilities [14] in the Linux kernel, we also found that at least 34% of them are related to incorrect error handling. Critically, erroneous error handling may result in many security issues such as use-after-free [15], information leakage [16], and denial-of-service [17].

The error-prone nature of error-handling code stems from several reasons. First, the error-handling code often deals with corner cases that are less likely to occur during normal execution. This results in two problems: Bugs in the error-handling code are

often not triggered or noticed, and developers tend to overlook such rare cases. We argue that, in adversarial scenarios, attackers can intentionally trigger error-handling code through techniques like memory exhaustion [18] and fault injection [19]. Thus the bugs can be equally critical to the ones in normal code. Second, traditional dynamic testing, such as fuzzing, cannot adequately cover the majority of error-handling code because errors are hard to trigger in fuzzing. Third, error handling often involves special and complicated logic, which poses significant challenges to correct implementation.

Developers in low-level languages mainly use four error-handling primitives. (1) Terminating execution. When an error is critical, the error handling terminates the execution to avoid attacks or data/file corruption. (2) Printing error messages. The code prints out the details about the error for users to investigate further. In this case, the error is less critical, so that the execution can continue. (3) Passing error upstream. The function encountering the error passes the error back to the callers and expects the callers to handle it further. (4) Fixing errors. When the error is fixable, the error handling can directly fix it (e.g., resetting the size value) and continue.

Prior research thus has primarily focused on detecting bugs only in the "handling." For example, Rubio-González et al. [20] and EIO [13] proposed static-analysis approaches to detect error-propagation bugs in file systems, i.e., if error codes are passed correctly. APEx [21], ErrDoc [22], and EPEx [23] also check if errors are identified and handled in the callers. Although a few previous works attempted to check the operations before the handling, they are limited to only missing-operation cases. For example, Hector [11] detects missing memory release, and RID [12] detects missing refcount decrease. To the best of our knowledge, none of them could detect other bugs associated with the operations in error paths before handling, such as cases in which the operations are present but in an incorrect order or redundant. To fill this blank area, in this work, we aim to systematically study and detect the bugs of problematic operations in error paths.

2.1 General rules for reducing the reliance on functionality semantics

In this section, we discuss the unique structures of error handling and present our study of DiEH.

```
/* drivers/media/platform/s5p-g2d/g2d.c */
static int g2d_probe(struct platform_device *pdev) {
    ...
    ret = v4l2_device_register(&pdev->dev, &dev->v4l2_dev);
    if (ret)
        goto unprep_clk_gate;
    vfd = video_device_alloc();
    if (!vfd) {
        ret = -ENOMEM;
        goto unreg_v4l2_dev;
    }
    ...
    ret = video_register_device(vfd, VFL_TYPE_VIDEO, 0);
    if (ret)
        goto rel_vdev;
    ...
    dev->m2m_dev = v4l2_m2m_init(&g2d_m2m_ops);
    if (IS_ERR(dev->m2m_dev))
        goto unreg_video_dev;
    ..
unreg_video_dev:
    video_unregister_device(dev->vfd);
rel_vdev:
    video_device_release(vfd);
unreg_v4l2_dev:
    v4l2_device_unregister(&dev->v4l2_dev);
unprep_clk_gate:
    ...
}
```

Figure 2.1: Example of the error-handling structure.

2.1.1 Error handling and function pairs

In case of an error, functions usually first clean up or handle the previous operations, e.g., releasing memory, before actually handling the error (e.g., returning an error code to their callers). Unwinding previous operations is however error-prone. To understand the characteristics of the handling of previous operations, we introduce the idea of leader and follower functions and use an example to describe the error-handling structure.

Leader and follower functions. Resources such as memory and locks are limited. As such, an operation against a resource, such as memory allocation, is typically accompanied by another operation that balances or recovers the resource. We define a function as a leader function if it initiates an operation against a resource. The

operation typically either acquires or changes the state of the resource. Correspondingly, we define a function as a follower function if it recovers the resource. The leader function and the corresponding follower function constitute a *function pair*. Common function pairs include allocation/deallocation, lock/unlock, refcount increase/decrease, etc. As an example, Figure 2.1 shows three pairs of functions. The first pair is v4l2_device_register() and v4l2_device_unregister(), which initializes and cleans up the related objects such as refcounts and locks. The second pair is video_device_alloc() and video_device_release(), which allocates and releases the memory for video devices. The third pair is video_register_device() and video_unregister_device() whose functionality is similar to the first function pair.

Unique error-handling structure—EH stacks and deltas. We identify a unique and common error-handling structure and refer to it as *EH stacks and deltas*. We use the example in Figure 2.1 to illustrate the structure. In the error paths, follower functions are called to handle leader functions in a "stack" manner (i.e., the last follower corresponds to the first leader). In EH stacks, we use unfilled circles to represent the functions in the normal paths, gray-filled circles to show the functions in the error paths, and black-filled circles to indicate the errors or error checks. In the example, v4l2_device_register(), video_device_alloc(), and video_register_device() are leader functions and are called sequentially: ④–⑦–⑬. In case of an error in v4l2_m2m_init(), ⓱, the error path is ㉒–㉔–㉖. In the path, the corresponding follower functions are called in reverse order, hence we call the structure *EH stack*. Due to the complexity of error handling and the poor design of certain follower functions, in practice, the structure may not be honored, leading to DiEH.

In this example, there are multiple EH stacks, two of which are: ④–⑦–㉖ (path 1) and ④–⑦–⑬–㉔–㉖ (path 2). When we compare the unfilled lines and gray-filled lines in these two EH stacks, we can obtain the difference which is ⑦–㉔. We call the difference an *EH delta*. In this particular case, the delta consists of only one leader function and one follower function. As such, we can infer that functions video_device_alloc() and video_device_release() are a *function pair*. The inference does not require any domain knowledge or understanding semantic structure, thus it can be automated. In HERO, we will leverage EH stacks and deltas to precisely pair functions.

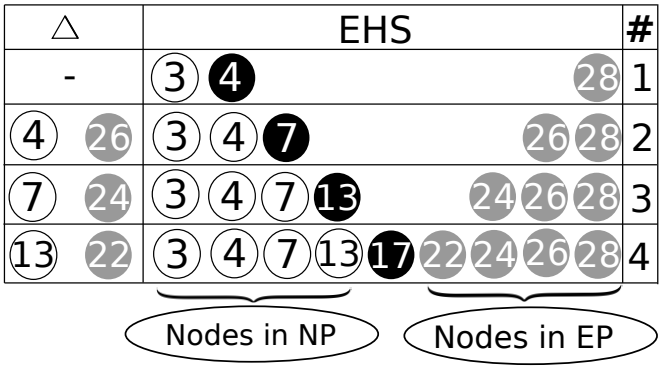

Figure 2.2: The EH stacks of the function in Figure 2.1. EHS = EH stack, EP = error path, NP = normal path, Δ = the EH delta of EHS_i and EHS_i-1 where the EH stacks are numbered in the "#" column.

2.1.2 Disordered Error Handling

In this subsection, we present the definition, categorization, causes, and security impacts of DiEH.

2.1.3 Definition of DiEH

DiEH represents cases in which the follower functions are called in an incorrect order, redundantly, or inadequately. Thus, a DiEH case occurs if it satisfies the three conditions: (1) a function contains at least one error paths, (2) the function has at least one leader functions, and (3) in some error paths, the corresponding follower functions are not called in order, exactly once, or adequately. Informally, we define a DiEH case as follows.

Definition 1 *Let EP be an error path in a function, [LD] be the list of leader functions in EP, [FL] be the actual list of follower functions in EP. Suppose [FL]′ is the expected list of follower functions to appear in EP based on the foreknowledge of function pairs, then:*

$\exists DiEH \in EP$, *if [FL]* $\neq$ *[FL]′*

Specifically, [FL] $\neq$ [FL]′ can occur due to three situations. (1) [FL] and [FL]′ contain the same set of follower functions but in different orders. (2) One or more follower functions are in [FL]′ but not in [FL]. (3) One or more follower functions are in [FL] but not in [FL]′. Based on the definition, we identify the key challenge in detecting DiEH as

collecting $[\mathrm{FL}]'$, which requires the foreknowledge of function pairs. In §2.3, we describe our new technique, which precisely identifies function pairs.

```
/* drivers/media/platform/rockchip/rga/rga.c */
static int rga_probe(struct platform_device *pdev) {
  ...
  pm_runtime_enable(rga->dev);
  ...
  ret = v4l2_device_register(&pdev->dev, &rga->v4l2_dev);
  if (ret)
    goto err_put_clk;
  vfd = video_device_alloc();
  if (!vfd) {
    ...
    goto unreg_v4l2_dev;
  }
  ...
  rga->vfd = vfd;
  ...
  rga->m2m_dev = v4l2_m2m_init(&rga_m2m_ops);
  if (IS_ERR(rga->m2m_dev)) {
    ...
    goto unreg_video_dev;
  }
  ...
  /* Create CMD buffer */
  rga->cmdbuf_virt = dma_alloc_attrs(...);
  rga->src_mmu_pages = (unsigned int *)__get_free_pages(...);
  rga->dst_mmu_pages = (unsigned int *)__get_free_pages(...);

  ret = video_register_device(vfd, VFL_TYPE_VIDEO, -1);
  if (ret) {
    v4l2_err(&rga->v4l2_dev, "Failed to ...");
    goto rel_vdev;
  }
  ...
  return 0;

rel_vdev:
    video_device_release(vfd);
unreg_video_dev:
    video_unregister_device(rga->vfd);
unreg_v4l2_dev:
    v4l2_device_unregister(&rga->v4l2_dev);
err_put_clk:
    pm_runtime_disable(rga->dev);
    return ret;
}
```

Figure 2.3: An example showing various new DiEH bugs, found by HERO, in a single function in the Linux kernel.

2.1.4 Classification of DiEH bugs

In §2.1.3, we present three situations that result in [FL] $\neq$ [FL]$'$. In this section, we present concrete cases for them.

Incorrect-order follower functions. Using correct follower functions but in an incorrect order can cause security bugs. For example, Figure 2.3 contains a use-after-free bug caused by using the follower functions in an incorrect order. The function video_device_alloc() is called in line 9, which is before the function call of video_register_device() in line 28. Thus, the corresponding follower release function, video_device_release(), should be called after video_unregister_device(). However, the error path starting from line 31 calls video_device_release() before calling the function video_unregister_device(). This incorrect-order DiEH results in a use-after-free because rga->vfd is an alias of vfd (line 15), and line 39 uses rga->vfd which uses the memory freed by line 37.

Redundant follower functions. Follower functions of a leader function might be called redundantly. This can happen when either multiple follower functions are called by mistake, or a follower can actually correspond to multiple leader functions, which confuses developers. For example, in Figure 2.3, the follower function video_unregister_device() (line 39) is called even when the call of its leader function video_register_device() (line 28) returns an error, which is unnecessary, leading to a redundant DiEH bug. A correct case is to call video_unregister_device() *only* when its leader function video_register_device() succeeds. Common issues resulting from redundant DiEH include double free, double unlock, double refcount, etc.

Inadequate follower functions. This situation refers to that necessary follower functions are missing. Common cases include missing release, missing unlock, missing refcount decrease, etc. For example, Figure 2.3 also contains several missing-release bugs. When the call of the function video_register_device() failed (line 28), pointers rga->cmdbuf_virt, rga->src_mmu_pages, and rga->dst_mmu_pages are not released, which are allocated in lines 24, 25, and 26. These bugs are common, and the Linux kernel has more than two thousand patches to fix inadequate follower functions. Prior research has studied such inadequate follower functions like missing resource release [11]; however, the other two situations, incorrect-order and redundant follower functions

remain unexplored.

2.1.5 Causes of DiEH

In this section, we summarize three major causes of DiEH based on our empirical analysis, which are hard to avoid.

Poor design of follower functions. Different programmers have various programming habits. Some follower functions are hard to use if they do not follow the programming convention. For example, functions pm_runtime_get_sync() and kobject_init_and_add() are called many times in the Linux kernel, but they are actually poorly designed. Both of these functions would increase the kernel refcount, even when they failed, violating good design practice. Some Linux maintainers we interacted with even complained that *"if you follow the common convention, you will get it wrong."* Though patterns and anti-patterns in API design are widely discussed [24], factors such as a need for backward compatibility and a large developer base makes API design a challenge.

Complexity and dependency of cleanup operations. Error paths are prevalent in a large program, and each may contain various cleanup operations (follower functions). Our analysis shows that, in the Linux kernel, there are more than 120K intra-procedural error paths, and 61.6% of them include at least one follower function, and on average, there are 2.46 follower functions per error path. The most complex error path contains 143 follower functions. More critically, some follower functions are dependent on each other, e.g., a parameter of a memory-release function is a nested field of a parameter of another function. The dependency requires the follower functions to be called in a specific order.

Custom follower functions. Different modules employ different leader and follower functions. We determined that about 80% of function pairs in the Linux kernel are custom (§2.5.1). These function pairs are defined and used within a specific module such as a driver. Avoiding DiEH bugs requires programmers to be knowledgeable about all the custom functions, which is a burden.

2.1.6 Prevalence of DiEH

It is hard to avoid DiEH due to the causes mentioned in §2.1.5. After manually checking 100 CVE-assigned vulnerabilities in 2019 from the Linux kernel, we found that DiEH causes 22 of them. Further, after checking the patches over the past two years from the Linux kernel, we found 42% of memory leaks and 45% of double-free bugs are due to DiEH. These results indicate that DiEH bugs are prevalent in the OS kernels, and can cause a wide range of security impacts. By employing a systematic detection, we expect to find many DiEH bugs.

2.1.7 Security Impacts of DiEH

Most DiEH bugs can cause severe security impacts, depending on their contexts. Common security impacts of DiEH include use-after-free, double-free, NULL-pointer dereference, deadlock, memory leak, refcount leak, etc. In the following, we showcase how DiEH leads to critical security issues.

Memory corruption. DiEH bugs often cause critical memory corruption such as use-after-free, double free, and NULL-pointer dereference. In Figure 2.3, we have shown how an incorrect-order DiEH leads to a use-after-free. Also, redundant and inadequate DiEH can lead to memory corruption. For example, CVE-2019-15504 [25] is a double-free vulnerability in the Linux kernel caused by redundant DiEH. This vulnerability has the highest CVSS score (10), which may be exploited remotely to compromise the system. CVE-2019-15292 [26] is a use-after-free vulnerability in the Linux kernel caused by inadequate DiEH. This vulnerability also has the highest CVSS score (10), which can compromise the confidentiality, integrity, and availability of the system. Further, DiEH is a source for NULL-pointer dereference. For example, CVE-2019-15923 [27] is a NULL-pointer dereference vulnerability in the Linux kernel, which is caused by inadequate DiEH.

Privilege escalation. DiEH can cause privilege escalation, which is considered one of the most critical security problems. CVE-2019-5607 [28] and CVE-2016-0728 [29] are refcount-leak bugs found in FreeBSD and the Linux kernel. Both bugs can cause privilege escalation because an overflowing reference count triggers a use-after-free. Similarly, CVE-2019-0685 [30, 31] is a refcount-leak vulnerability in Windows, which can be exploited to

launch privilege-elevation attacks. These results show that DiEH can also compromise the confidentiality and integrity of OS systems.

Denial-of-Service. The most common security impact of inadequate follower functions is resource leak such as memory leak and refcount leak. Memory leaks in the OS kernels are considered critical because they can crash the whole system and lead to Denial-of-Service (DoS) [32, 33]. Figure 2.3 is vulnerable to a memory leak in case video_register_device() fails.

2.2 Overview of HERO

Based on the study, we develop an effective detection system for DiEH bugs. In this section, we first discuss the challenges in identifying DiEH, and then outline our solutions.

2.2.1 Challenges in Identifying DiEH

While prior research [12, 11] attempted to detect cases of inadequate follower functions, cases of incorrect-order and redundant follower functions remain unexplored. Systematically detecting DiEH bugs involves three major challenges.

Analysis of error-handling structures. HERO first needs to analyze the error-handling structures, so as to extract EH stacks and deltas, which will be leveraged to identify function pairs. In particular, HERO needs to: (1) identify the normal paths (e.g., ④, ⑦, and ⑬ in Figure 2.1) and error paths (e.g., 22, 24, and 26 in Figure 2.1) in a function, (2) identify the leader and follower functions in the normal and error paths.

Normal and error paths are often interleaved in the program. Thus, to identify and distinguish them, we need to know their demarcation points, which is a non-trivial task. In a function, there may be many normal and error paths, but only some of them should be associated together. Thus, we should map the normal paths to their corresponding error-paths. Further, numerous functions are called in normal and error paths, but not all of them should be called in pairs. Therefore, we need to extract the leader functions from normal paths and follower functions from the corresponding error paths. More importantly, as we will describe in §2.3, the pairing of a leader and a follower function can be either conditional or unconditional. A precise pairing analysis requires distinguishing them, which is hard.

Function-pair identification. According to our definition of DiEH (§2.1.3), the detection of DiEH is essentially checking [FL] $\neq$ [FL]$'$, which requires the foreknowledge of leader-follower function pairs. This would previously require domain knowledge or manual efforts, and its automation is a significant challenge. In particular, programs tend to extensively use custom functions—defined and used in a specific module. Such functions have a limited number of uses, so existing mining-based inferences may not work. In fact, our study estimates that 80% of follower functions in the Linux kernel are custom. Moreover, there are a number of different classes of leader-follower pairs, such as allocation/deallocation, lock/unlock, getter/putter, and register/unregister. As a result, previous works (e.g., [34, 35]) either assume that function pairs are provided or only target a specific class of common pairs.

Elimination of harmless DiEH cases. The checking of "[FL] $\neq$ [FL]$'$" returns DiEH cases which may not be harmful, i.e., false positives. There are two major causes of harmless DiEH cases. First, by nature, the path of an EH stack often involves path constraints (i.e., return-value check). A path is infeasible if conflicting constraints exist. The intuitive solution, symbolic execution, may not work in complex programs. Second, for the incorrect-order DiEH cases if the follower functions are independent, their order does not matter. Therefore, for these incorrect order DiEH cases, we need to understand the potentially complicated data dependencies among different follower functions to determine potential bugs. Note that redundant and inadequate DiEH cases do not have this challenge because they are independent of ordering.

2.2.2 HERO Techniques

To address the challenges, we propose multiple new techniques. In this section, we briefly introduce them.

Understanding error-handling structures. To identify function pairs, HERO first automatically understands the error-handling structures and represents them with a graph. This technique starts with identifying *error checks*. An error check is basically an if statement that checks whether a function returns an error code. For example, lines 5, 8, 14, and 15 are error checks in Figure 2.1. With the identified error checks, we can identify normal paths and error paths—the code path prior to the error check is the

normal path, while the taken path (as opposed to the fall-through) of the error check is the error path. This technique also identifies leader and follower functions on the normal and error paths by removing irrelevant functions (e.g., via dependency analysis), and stores all the information in a graph, referred to as the error-handling graph or *EHG*. We will present details of the technique in §2.3.1.

Pairing functions with EH deltas. To identify function pairs in a program, we propose delta-based analysis, which can precisely pair functions even when they are custom (i.e., only with a small number of occurrences). The key insight is that follower functions in the error path are called in a specific (reverse) order, corresponding to the leader functions in the normal path, which constitutes EH stacks, as shown in Figure 2.2. More importantly, when we compare two *adjacent* EH stacks, we naturally obtain the EH delta, which oftentimes has only one leader function and one follower function—therefore, we can infer that this follower function is paired to the specific leader function. For example, by comparing EH stacks 1 and 2 in Figure 2.2, we obtain the EH delta, ④–㉖, which constitutes a function pair. Similarly, EH stacks 2 and 3 generate the EH delta, ⑦–㉔, forming another function pair. To further improve the pairing precision, we propose *EH-stack calibration* to distinguish conditional and unconditional pairs. Details are presented in §2.3.

Detecting DiEH bugs with identified pairs. To detect DiEH bugs, we first detect DiEH cases, and then remove infeasible and harmless cases to report DiEH bugs. HERO detects DiEH cases by comparing the follower function list [FL] with the expected follower function list [FL]′.

To remove infeasible DiEH cases, we propose a scalable *symbolic summary* for conflicting constraints, which helps eliminate infeasible paths. In addition, to remove harmless incorrect-order DiEH cases, we propose *follower-dependency reasoning*, which finds independent follower functions whose order does not matter. Finally, we provide a ranking of detected DiEH bugs to facilitate manual confirmation. More design details will appear in §2.3.3.

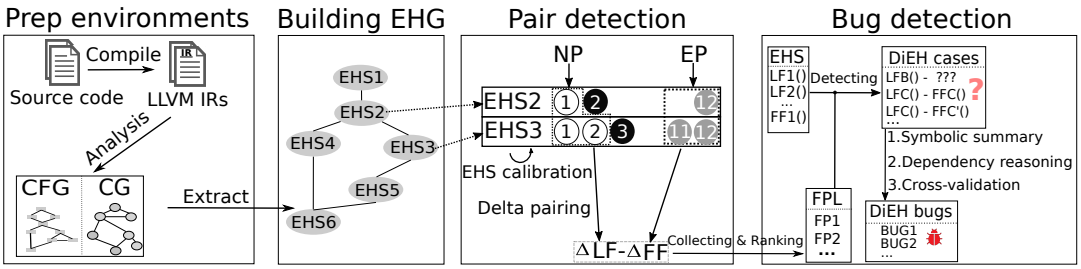

Figure 2.4: An overview of HERO. It has four steps; by taking the source code of a program, it automatically reports ranked DiEH bugs. CFG = Control flow graph, CG = call graph, EHG = error-handling graph, EHS = error-handling stack, LF = leader function, FF = follower function, FPL = function-pair list, Sym sum = symbolic summary.

2.2.3 The HERO Framework

We now briefly introduce the workflow of HERO, shown in Figure 2.4. HERO consists of four steps. (1) *Preparing the analysis environment.* HERO first prepares the analysis environments by compiling the source code to LLVM IRs (bitcode files), and building the control-flow graph (CFG) and call-graph (CG) for the program. (2) *Constructing error-handling graph.* Second, HERO analyzes the unique error-handling structures to extract errors and EH stacks for each function. After that, the HERO constructs an EHG to record all the information. (3) *Leader-follower pairing.* Third, based on the EH stacks, the HERO computes the EH deltas and leverages them to pair functions. (4) *DiEH detection.* Finally, based on function pairs and the EHG, HERO detects DiEH bugs in the program. As a result, HERO reports the DiEH bugs. The reports include details such as disordered situations and suggested fixes.

2.3 Design of HERO

A key challenge to detect DiEH bugs involves identifying function pairs including custom ones. We propose a novel technique that leverages the unique error-handling structure—EH stacks and deltas—to precisely pair functions. In this section, we present the design of the pairing analysis.

2.3.1 Extracting Error-Handling Structures

Identifying error checks, normal and error paths. To extract EH stacks of a function, we first identify error checks to collect normal paths and error paths. To identify error checks, we collect common error codes such as ENOMEM, and common error-handling functions such as pr_err() and panic(); §2.4 presents more details. Such error codes and error-handling functions are typically uniformly defined in dedicated header files. HERO regards a path as an error path if it returns an error code or a NULL pointer, or calls at least one error-handling functions. This design is consistent with existing works [21, 23, 11, 36].With the identified error checks, we naturally collect both normal paths and error paths of each error check. A path is represented with a list of code blocks, and a function can potentially contain a large number of paths.

Filtering follower functions by removing noises. Not all the functions in the normal and error paths should be paired, e.g., kprintf(). Therefore, we want to remove irrelevant functions. We first remove noisy functions in the error paths, i.e., filtering follower functions. We observe that unlike normal paths, error paths tend to be much simpler, in which irrelevant functions are typically commonly used error-messaging (e.g., dev_err()) and exiting (e.g., panic()) functions. Therefore, we remove such functions, and details are presented in §2.4.

Filtering leader functions through data dependency. Compared to error paths, normal paths are more complicated, which call diverse functions. As such, we instead employ data dependency to filter potential leader functions, given that we have already selected potential follower functions mentioned above. The insight is that follower functions clean up resources obtained by or operations performed by leader functions; a leader function and the corresponding follower function should be connected through variables. For example, kfree() takes the pointer returned by kmalloc() as the parameter. With the insight, we select potential leader functions based on data dependencies on the selected follower functions. Specifically, if the return value or a parameter of a function is used by a follower function, we select it as a potential leader function. To be conservative, our dependency analysis is field-insensitive. That is, different fields of an object are also considered dependent.

Constructing EH stacks. With the potential leader and follower functions collected,

we next construct EH stacks. An EH stack consists of three parts: <ERROR, [LD], [FL]>. Here, [LD] is a non-empty list of leader functions, which are in the normal path; [FL] is a non-empty list of follower functions, which are in the error path; ERROR is the call-site to the error-generating function corresponding to the error check. We bypass the path-explosion problem by collecting EH stacks using intra-procedural analysis.

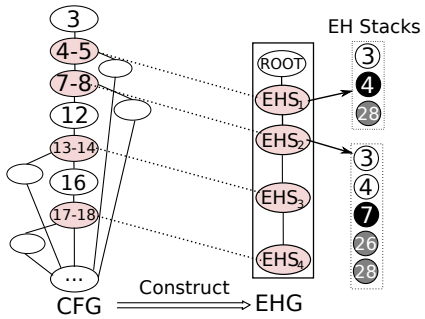

Figure 2.5: Constructing the EHG for the function in Figure 2.1. EHS = EH stack. With the EHG, we can quickly find adjacent EH stacks to compute EH deltas.

Building error-handling graph. To record all the identified error-handling information for a function, we then build an error-handling graph (EHG). Another purpose of building EHG is to also capture the adjacency of EH stacks, which facilitates the pairing analysis. The nodes of the EHG are EH stacks. Edges are added to connect the EH stacks based on the control-flow dependencies of the error checks associated with the EH stacks.

Specifically, given a function, to build the EHG, HERO first constructs the nodes by identifying all the basic blocks that include an error check. Then from a selected basic block and its error check, HERO collects all the EH stacks associated with this error, and further records these EH stacks into the nodes of EHG. After that, HERO traverses the CFG and connects these nodes in the EHG based on their control-flow relationship. Figure 2.5 shows an example of creating the EHG based on the control-flow graph (CFG) of the function in Figure 2.1. Four shadow nodes, which mark lines 4-5, 7-8, 13-14, and 17-18 in the CFG, indicate the code blocks containing error checks.

2.3.2 Delta-Based Pairing Analysis

In §2.3.1, we extract EH stacks and build the EHG. In this section, we present how we perform the delta-based pairing analysis, which computes EH deltas by comparing two adjacent EH stacks to precisely identify function pairs.

Computing EH deltas. As already described in §2.2.2, we leverage EH deltas to precisely identify function pairs because EH deltas often precisely capture an extra leader function and the extra follower function. To compute the EH deltas, we pick each two adjacent EH stacks from the EHG and compare them to generate the delta. In less than 5% of cases, an EH delta contains more than one leader or follower functions; in this case, we still try to pair them but in reverse order with the help of data-dependency analysis. That is, for the last follower function, it will be paired to the first leader function if they have data dependencies; otherwise, we would try to pair it with the second leader function. Following this order, and if finally, this follower cannot be paired with any leader function, we would further calibrate the EH stack (shown in the next paragraph) and try to pair it with the error-generating function. HERO would drop the leader or follower functions if they eventually cannot be paired, which is uncommon. Note that HERO may pair one leader to multiple follower functions, and vice versa, which means that the pairing output is "many-to-many" mapping between leader and follower functions.

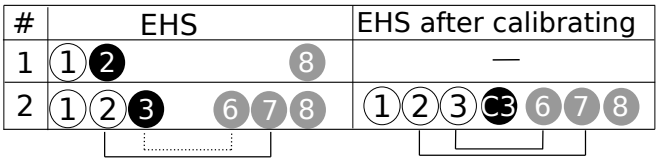

Figure 2.6: Calibrating EH stack. EHS = EH stack.

Calibrating EH stacks. Before we present the pairing algorithm, we first describe the challenge. We divide function pairs into two categories – *conditional pair* and *unconditional pair*. In most cases, function pairs are conditional. That is, a follower function is necessary only when the leader function succeeds. For example, if kmalloc() fails, kfree() is unnecessary. However, there are also some unconditional pairs. That is, despite the failure of the leader function, the corresponding follower function is still required. For example, as mentioned in [37], when kobject_init_and_add() fails,

its follower function kobject_put() is still required to clean up the related objects. To correctly construct EH stacks, we must distinguish conditional and unconditional function pairs; otherwise, the pairing results would be unreliable.

The pairing algorithm. Putting the steps together, HERO first traverses the EHG to get each EH stack and its successor in the EHG; these are two adjacent EH stacks. Specifically, HERO analyzes every path and differentiates error paths from normal ones to collect adjacent EH stacks. As such, HERO handles conditionals—if there is a conditional statement, HERO will simply collect two paths. Then, HERO calculates their EH delta. If the EH delta indicates an unconditional pair, HERO calibrates the EH stack and re-calculates the EH delta. Using the EH deltas, HERO collects the function pairs. The output of this algorithm is a list of potential function pairs. Note that this algorithm also includes a ranking mechanism that will be presented in §2.4.

2.3.3 Detection of Disordered Error Handling

With the identified function pairs and constructed EHG, HERO automatically detects DiEH bugs. The detection works with two phases: detecting DiEH cases, and reporting DiEH bugs by removing infeasible and harmless cases.

Detecting DiEH cases. HERO employs an intra-procedural, flow-sensitive static analysis to check each path and its corresponding EH stack in functions. At a high level, each EH stack contains a list of leader functions [LD] as well as a list of follower functions [FL]; after that, HERO computes the expected list of follower functions $[FL]'$ and compares it with [FL]. HERO reports cases in which $[FL] \neq [FL]'$ as DiEH cases. HERO also categorizes the DiEH into incorrect-order, redundant, and inadequate cases based on the classification rules presented in §2.1.5.

2.3.4 From DiEH Cases to DiEH Bugs

A DiEH case can be infeasible or harmless. In this section, we present our techniques for eliminating such cases to confirm DiEH bugs. We also provide a ranking mechanism to prioritize DiEH cases.

Eliminating infeasible paths by detecting conflicts. HERO statically finds normal and error paths to detect DiEH. If a path is infeasible (i.e., containing conflicting path

constraints), the detected DiEH would be a false positive. To remove such false positives, we aim to eliminate infeasible paths. An intuitive strategy is to employ traditional symbolic execution, which is not scalable and can easily lead to path explosion, not to mention that our target programs are complex. To address this problem, we propose a scalable *symbolic summary* for each function, which intra-procedurally captures *conflicting* constraints among the variables such as, conditional variables and return values. When a path contains such conflicting constraints, we deem it infeasible.

Specifically, the symbolic summary consists of two steps: (1) collecting constraints from the path under analysis, (2) checking the existence of conflicting constraints. In the first step, HERO analyzes the current path and collects constraints from every conditional statement, such as if (flag == True). Further, HERO extracts changes against the variables of collected constraints that we are certain about, such as constant assignment like flag = false. If a change is uncertain, e.g., assigned with an unknown variable, we regard the case as an uncertain constraint. In the second step, HERO checks collected constraints, and treats the path as infeasible if it has conflicting constraints. (e.g., the first constraint is flag == false and then the second constraint is flag == True.) The symbolic summary conservatively regards all the uncertain constraints as solvable, ensuring the precision of the removal of infeasible paths. This simple approach can quickly and reliably (i.e., the infeasibility is determined) remove infeasible paths without handling complicated uncertain constraints, which is a lightweight version of under-constrained symbolic execution.

Figure 2.7 shows an example of conflicting constraints causing a false positive in detecting DiEH. For this case, without the symbolic summary, HERO would detect a missing-follower DiEH case—the release function, kfree(max3421_hcd->rx), is missing in path ③–④–⑦–⑪–⑫–⑬–⑮–⑯–⑲. This is however a false positive because constraints if(!hcd) (line 4) and if(hcd) (line 16) are conflicting in the path. With the symbolic summary, when analyzing this path, HERO will first collect the constraint hcd != NULL from line 4 and the constraint hcd == NULL from line 16. Then, HERO determines that the constraints are conflicting, and thus the path is infeasible. In addition to checking conflicting constraints from a called function, our technique will check the ones from the current function and use them to eliminate infeasible paths. To collect more conflicting constraints, we also employ alias analysis, which is based on the LLVM

alias analysis infrastructure [38] to map the variables involved in the constraints.

```
/* drivers/usb/host/max3421-hcd.c */
static int max3421_probe(struct spi_device *spi) {
    hcd = usb_create_hcd( ... );
    if (!hcd)
        goto error;
    ...
    max3421_hcd->rx = kmalloc( ... );
    if (!max3421_hcd->rx)
        goto error;

    max3421_hcd->spi_thread = kthread_run(...);
    if (max3421_hcd->spi_thread == ERR_PTR(-ENOMEM))
        goto error;
    ...
error:
    if (hcd)
        kfree(max3421_hcd->rx);
    ...
    return retval;
}
```

Figure 2.7: Example of the conflicting constraints.

Our evaluation shows that our solution is effective, and it reduces about half of the false positives cases without introducing additional false negatives, which makes the results manageable for manual analysis. Nevertheless, our symbolic summary is based on intraprocedural analysis and only considering the most intuitive conflict constraints, and thus it still cannot handle the false positives caused by complicated conditions. The evaluation results in §2.5.5 show that, finally, for bug detection, 23% of false positives are caused by complex conditions, which cannot be handled by the symbolic summary. However, our intra-procedural symbolic summary and feasibility testing are highly scalable, with no noticeable slowdown in the analysis.

In general, we can compare the symbolic summary with the symbolic execution from the following aspects: (1) both do not have false-positive in theory, (2) the symbolic summary has false-negatives due to the intraprocedural analysis and also missing handling complex constraints, and (3) the symbolic summary performance is much better than symbolic execution because the front one would not suffer from complex constraint solving or copying state for the forked process, which only simply compares the must conflict constraints in a given path.

Eliminating harmless cases via dependency reasoning. HERO reports any incorrect-order follower functions as potential DiEH. However, we observe that if two follower functions are independent, it is typically harmless to call them in staggered order. Therefore, we eliminate such independent cases. Specifically, we employ dependency reasoning to find independent follower functions. To be precise, we employ MustAlias analysis [38] and field-sensitive analysis. We apply the data-dependency analysis to the parameters and return values of the follower functions. If data dependency is found, we keep the DiEH cases. This technique can effectively remove the harmless incorrect-order DiEH cases.

Ranking reported bugs through cross-validation. To alleviate the manual effort in confirming DiEH bugs, HERO further ranks reported cases by employing cross-validation [39] across the cases. HERO calculates the percentage of error paths that encounter this problem. A lower percentage indicates that the DiEH case is an outlier and is more likely a bug. HERO then ranks the bugs based on the percentage in ascending order, for each category.

2.4 Implementation

We implement HERO based on LLVM-10 as multiple passes that identify error-handling structures, construct the EHG, perform delta-based pairing analysis, and detect DiEH bugs. We also implement multiple Python scripts for pairing and bug ranking. HERO is implemented with 5.5K lines of code in C++ and 800 lines of code in Python. In this section, we present some interesting implementation details.

Removing irrelevant functions in error paths. Compared to normal paths, error paths are often simple. Typically, irrelevant functions can be either (1) error-logging functions (e.g., dev_err), which log error messages, or (2) exit functions (panic), which terminate the execution. We employ two methods to eliminate such functions. First, we find that error-logging functions have clear patterns, e.g., having variadic and format parameters. We identify such functions by using pattern-matching. Second, to collect terminating functions, we identify wrapper functions that internally call primitive ones like panic(), abort(), and exit(). In total, we collect 537 irrelevant functions that are excluded from the pairing.

Ranking function pairs. The pairing analysis is precise for most cases but still has some false positives (see Figure 2.8) due to limitations with static analysis. We thus also provide a ranking mechanism against the pairs. The key insight is that for a true function pair, the occurrences of the leader function should close to the occurrences of the follower function. Given a function pair, we count the total occurrences of a leader function as LT and the total occurrences of its follower function as FT. Then, we count the frequency of function pair occurrence in the program as PT. Finally, we define the *paired rate (PR)* as $PR = \frac{PT^2}{FT*LT}$ and use it to rank the pairs in descending order. If PR approaches one the leader function and follower function are always used together; on the other hand, if PR approaches zero, the leader and the follower are rarely paired. Our evaluation (see §2.5.2) shows that such ranking can effectively squeeze most of the false positives into the bottom of the list, which can be eliminated easily.

2.5 Evaluation

We conduct our experiments on an Intel Xeon CPU server that has 48-cores and 256GB RAM, and runs Ubuntu-18.04 OS. All experiments use -O2 optimization to generate bitcode (LLVM IR) files. We evaluate HERO on both system and application software, including Linux (commit #: 4d856f72c10) and FreeBSD (commit #: c54c07625bd) kernels, and OpenSSL library (commit #: 7821585206).

Analysis time and program complexity. Table 2.1 shows the analysis time for each component across different systems. Even for the Linux kernel, which has 17.7 million lines of code, the pairing finishes within one hour, and the detection finishes in about 10 hours. The results confirm that HERO is efficient and can scale to large programs. Note that HERO is currently single-threaded; multithreading can further improve its efficiency.

Target program	**Lines of Code**	**IR files**	**Time for pairing**	**Detection time**
Linux kernel v5.3	17.7M	18,071	48 min	10 h 16 min
FreeBSD v12.1	4.8M	1,483	10 min	2 h 28 min
OpenSSL	450K	1,902	53 sec	11min

Table 2.1: Analysis time of HERO and the complexity of programs.

Preparing pair sets. To evaluate our delta-based pairing, we prepare two sets of function pairs. The first set is the *reported pair set*, which includes 150 randomly selected unranked functions pairs identified by HERO. As will be detailed in §2.5.2, 89 of them are true pairs, while 61 are false pairs. The second set is the *ground-truth pair set*, which includes 86 function pairs of various types. We collected this set from 15 random source files across different subsystems of the Linux kernel; these files contain 26K lines of source code.

2.5.1 Characteristics of Identified Pairs

HERO detects more than 7.5K, 416, and 323 potential function pairs in the Linux kernel, OpenSSL, and FreeBSD, respectively. To further characterize these pairs, we pick the Linux kernel because it is the most complex. We first use script code to statistically select common keywords in the names of paired functions, and use the keywords to empirically classify pairs. The common keywords and the classification are summarized in Table 2.2. Interestingly, the keywords of a pair usually have the opposite meaning, indicating the paired operations, e.g., alloc/dealloc and increase/decrease.

Classes (Proportion)	**LF Operations**	**FF Operations**
Resource acquisition (50.2%)	alloc, new, request, create init	free, release, erase, destroy, remove fini, finish, deinit, uninit
Lock (4.4%)	lock, down	unlock, up
Refcount (12.5%)	get, inc	put, dec
Device related (18.2%)	register charge, on, enable	unregister, deregister uncharge, off, disable
Bit operation (0.7%)	set	clear
Others (33%)	apply, pin, assert join, add, map reserve begin, start, open setup	revert, unpin, deassert leave, remove, unmap delete, del end, finish, stop, exit, close clean, cleanup

Table 2.2: Common classes of function pairs in the Linux kernel. LF and FF are leader and follower functions, respectively.

Custom function pairs. A strength of our delta-based pairing analysis is that it does not require a large number of occurrences of pairs for inference or mining. As a result, HERO is capable of identifying function pairs that are composed of custom leader and follower functions, and thus it can identify a significantly larger number of pairs.

To confirm that, we identify custom pairs from the 89 true pairs in the aforementioned reported pair set. We find that 71 are defined and used in specific modules, thus are custom. Therefore, the result shows that 79.8% of them are custom.

2.5.2 Precision and Recall of Delta-Based Pairing

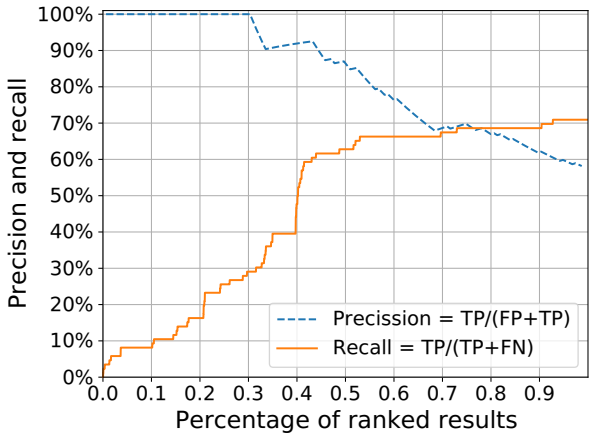

Figure 2.8: Precision rate and recall rate of pairing results. TP = True positives, FP = False positives; FN = False negatives.

Precision of the pairing. We first evaluate the precision rate (i.e., TP / (FP + TP)) of the pairing and the ranking mechanism. Manually confirming all the detected pairs is impractical, so we reuse the "reported pair set" which contains 150 unranked function pairs. In particular, we manually confirm the pairs through their names, semantics, functionalities, and usage by reading the code and comments. We found that functions names are very helpful in the confirmation because they contain opposite keywords (e.g., alloc/dealloc) and follow similar structures. Our confirmation shows that 89 are true function pairs. Automatically pairing the functions in large programs like Linux, where custom functions are prevalent, is very challenging. We believe the precision is already promising. However, to further improve the precision, we also provide a ranking mechanism, as shown in §2.4. The evaluation results show that the ranking mechanism can help exclude most of the false positives caused by irrelevant functions. As we discussed in §2.4, besides the 537 error-handling functions such as warn() , HERO treats all other functions as *potential* leader/follower functions. Thus other irrelevant functions can still incur noises. However, the false positives caused by these irrelevant

functions can be further filtered out by the "paired rate," which is based on the fact that the irrelevant functions are often not paired to its leader or follower functions. For example, the function __memcpy() is an irrelevant function for functions pairs, but HERO still paired __memcpy() and kfree() twice in the Linux kernel. Nevertheless, __memcpy() is called more than 27K times, and kfree() is called more than 30K times in the linux kernel. Thus, the paired rate for function __memcpy() and kfree() is nearly 0, which means that they are not really a function pair. Figure 2.8 shows the precision evaluation for ranked pairs: For the top 30% of ranked pairs, the precision is 100%, and even for the 75% of the ranked pairs, the precision is about 70%.

We summarize three major causes of false positives. First, irrelevant functions still exist in EH stacks, introducing noises in the pairing. Second, function pairs may not appear in the same function but across different functions. The current implementation of HERO employs an intra-procedural analysis which would miss such pairs. Third, the detection of error paths, which is based on error codes, may misidentify normal paths and error paths of custom error codes are involved, leading to false positives as well.

We also evaluate the recall rate (i.e., TP / (TP + FN)) of the pairing and the ranking mechanism using the aforementioned ground-truth pair set. The set contains 86 true function pairs; we find that HERO can detect 61 of them, leading to a recall rate of 0.71. Furthermore, Figure 2.8 shows the recall rate for the ranked pairs. Similar to the causes of false positives, false negatives are also mainly caused by (1) incorrect error-path identification and (2) noises in delta analysis.

Table 2.3: Comparison with the closest pairing tools PF-Miner [1] and PairMiner [2]: Number of function pairs per million lines of source code. The top 30% of pairs identified by HERO are precise.

	PF-Miner	**PairMiner**	**HERO**	**HERO** (30%)
Linux	-	94.7	303.3	128.2
Android	50.5	-	-	-

2.5.3 Comparison with Previous Pairing Analyses

We aim to compare HERO with related works on function pairing. We identify the following most relevant and recent works: PairMiner [2] and PF-Miner [1]. RID [12] also pairs functions; however, it focuses only on refcount-related ones and uses simple string

matching (e.g., *_inc/*_dec), so we exclude it from the comparison. PF-Miner [1] first employs string matching (e.g., new/delete and alloc/free) to collect functions. Then, equipped with a mining algorithm, it statistically pairs the functions that often show up pairwise in the normal and error paths. After analyzing the C source code of the Android kernel, PF-Miner identifies 546 paired functions. PairMiner [2] shares similar approaches because it is built on top of PF-Miner. PairMiner identifies 1023 paired functions in the Linux kernel.

We compare HERO with these tools in how many function pairs are identified. Unfortunately, we cannot compare precision values because neither tool provided such numbers. Note that PF-Miner and PairMiner both employ simple mining (i.e., statistical counting) to collect pairs, we believe they inherently suffer from precision issues and could not support custom functions. Table 2.3 presents the details of the comparison. Specifically, HERO is agnostic to types of function pairs and supports custom functions. HERO also identifies significantly more pairs. Even if we select the top 30% of ranked function pairs, the number is significantly higher compared to either PF-Miner or PairMiner. We attribute HERO's effectiveness to its delta-based pairing analysis, which is precise and can support custom functions.

Evaluation against dependency-based pairing. WYSIWIB [40] employs data dependencies to pair alloc/dealloc function pairs. To compare HERO to such pairing, we extend the dependency analysis to all functions in the normal and error paths. As a result, such pairing reports about 200% more function pairs; however, we found the majority of them are false positives (wrong pairs), disqualifying it for the DiEH detection. This result shows that delta-based analysis can significantly reduce false function pairs and make results more precise.

2.5.4 Bug Detection

Based on the precision and recall trade-off shown in Figure 2.8, we choose the top 43.2% of function pairs for detecting DiEH bugs because it achieves a high precision (92.5%) and a reasonably good recall (60.4%). We then apply HERO to three target programs, the Linux kernel, the FreeBSD kernel, and the OpenSSL library, with corresponding 3276, 94, and 123 function pairs detected.

Based on these function pairs, HERO finally identifies 234, 2, and 3 DiEH bugs from

the Linux kernel, FreeBSD, and OpenSSL library. The details of the identified bugs are shown in Appendix A. Among these detected DiEH bugs, 72% are caused by inadequate follower functions, 25% are caused by incorrect-order follower functions, and 3% are caused by redundant follower functions. Further, we found that the drivers of the Linux kernel are buggier than its core kernel. In the Linux kernel, the driver code accounts for 62% of the whole code-base; however, 87.6% of the found DiEH bugs come from the driver code, which means the bug density of the driver code vs. the core kernel is 4.3 : 1. We believe this is due to the following reasons: (1) drivers contain more custom functions, which are harder to be analyzed by previous static-analysis approaches; (2) many functions in drivers are used to support outdated devices and thus infrequently used or tested, and (3) compared to the core kernel, the drivers are less tested because existing dynamic-analysis tools require hardware devices or their emulation [41]. In the rest part of this section, we will present the causes of false positives and some interesting findings. For simplicity, we focus on the Linux kernel because it is the largest and the most complex.

2.5.5 False-Positive Analysis

HERO in total reports 454 potential DiEH cases in the Linux kernel, with 170 for incorrect-order, 40 for redundant, and 244 for inadequate DiEH cases. We manually check all these cases and regard a case as a true bug if it meets both of the following conditions: (1) the case is an actual DiEH case, and (2) the case would introduce at least one security issue. We confirmed 234 (thus, the false-positive rate is 48%) of them as true positives, with 58, 7, and 169 for incorrect-order, redundant, and inadequate DiEH bugs, respectively. To manually confirm these bugs, three researchers spent about a total of 16 man-hours. We believe the precision is reasonably good for static analysis–based detection against complex programs, and the manual effort for the confirmation is very manageable. Further, we patched and reported 230 bugs to the maintainers. The remaining 4 cases are removed in the latest version of the kernel. As of the submission of this paper, 125 of them have been accepted, and 105 have not received a response yet. We further analyzed the major causes of false positives.

First, we find that 23% of false positives are caused by complex path conditions that were missed by our under-constrained path-feasibility testing. We can mitigate these

false positives by collecting more constraints from the complex path conditions. Second, although some DiEH cases indeed exist, their impacts are prevented by some security operations such as enforcing a NULL check for a released pointer. Such cases contribute about 7% of false positives, and removing such false positives requires understanding the security operations. Third, our pairing analysis still misses the follower functions for some leader functions. This causes 18% of false positives. The remaining false positives are caused by other issues such as the aliasing problem in the static analysis, or incorrect detection of error paths.

2.5.6 Maintainer Feedback

During the bug confirmation and reporting, we found that function pairs are often used incorrectly. First, 8.2% of DiEH bugs are introduced by previous patches that incorrectly fixed error-handling bugs. For example, the patch (6e5da6f7d824 [42]) in the Linux kernel fixed a DiEH bug caused by inadequate follower function. However, when this patch calls function pm_runtime_get_sync(), it still misses pm_runtime_put() when the call of function pm_runtime_get_sync() fails, which results in the bug. Second, even experienced Linux maintainers are not familiar with some follower functions, particularly custom ones. For example, few maintainers were aware that kobject_put(P->kobj) releases pointers P and P->kobj. These results are consistent with our previous findings in §2.1.5—cleanup operations are common, complex, and difficult to get right.

2.5.7 Security Impact Analysis

We not only confirm DiEH bugs but also empirically determine the impact of confirmed bugs. The impact is based on the involved variables and the contexts of each bug. Our determination is conservative—if a case is too complicated to analyze, we exclude it from the bugs. We reported the rest of the bugs to maintainers.

We summarize the impacts of the confirmed bugs in Table 2.4. 98.0% of the bugs would cause at least one of the security impacts mentioned in the table. Specifically, 3.0% of DiEH bugs would lead to use-after-free, double-free, or double-unlock, and all of them are caused by redundant follower functions. As we discussed in §2.1.7, these DiEH bugs can lead to critical security issues like memory corruption, DoS, privilege escalation.

Table 2.4: Most common security impacts of bugs found by HERO. CWE = common weakness enumeration. IOF = incorrect order of follower function, IFL = Inadequate follower functions, RFL = redundant follower function.

Type of bugs	**Prop**	**Causes**	**CWE-ID [43]**
Refcount leak	85.8%	IFL (75.6%), IOF (24.4%)	CWE-911
Memory leak	9.2%	IFL (77.3%), IFO (22.7%)	CWE-401
UAF/DF	1.7%	RFL	CWE-416, CWE-415
Double unlock	1.3%	RFL	CWE-765

Further, 85.8% of DiEH bugs would lead to refcount leak, with 75.6% of them caused by inadequate DiEH and 24.4% caused by incorrect-order DiEH. People often regard refcount leaks as general bugs but not security-critical ones. However, we argue that refcount leaks can also cause memory corruption. When a refcount field, especially the one with only 16 or less bits, is repeatedly incremented, it will finally overflow to zero, triggering a free and finally causing a use-after-free. As we discuss in §2.1.7, CVE-2016-0728 [29] is such an example. Moreover, there are many examples of exploiting refcount leaks for privilege escalation (e.g., CVE-2016-0728, CVE-2014-2851) and DoS (e.g., CVE-2019-9857). DoS, like crashing in the kernel, is security-critical for long-running servers.

Also, 9.2% of DiEH bugs would lead to memory leaks, with 77.2% of them caused by inadequate DiEH and 22.7% caused by incorrect-order DiEH. Memory leaks in the kernel can also be critical because they may result in DoS of the whole system. Assigned CVEs of kernel memory leaks include CVE-2020-15393 [44], CVE-2019-8980 [45], CVE-2019-5023 [46].

Table 2.5: The numbers of DiEH bugs that can be triggered from system calls, ioctl handlers, and IRQ handlers.

Type of entry points	**Number of reachable bugs**
System calls	180 (76.9%)
ioctl handlers	190 (81.2%)
IRQ handlers	185 (79.1%)
Total	199 (85.0%)

Triggerability analysis for detected bugs. To further understand the security

impacts of bugs identified by HERO, we also tested the triggerability of them. Automatically confirming the triggerability of kernel bugs is still considered a challenging research problem. Dynamic analysis tools like OS fuzzers [47, 48, 49] have a low false-positive rate but suffer from performance issues and many false negatives. Therefore, similar to previous works such as SID [50], this evaluation focuses on identifying triggerable call stacks from the adversary-reachable entry points (e.g., system calls, ioctl handlers, and IRQs handlers) to the functions containing DiEH bugs. More details about the entry points are shown in Section VI.D of the SID paper [50]. Specifically, we analyze all the call instructions in the Linux kernel and leverage the state-of-the-art technique MLTA [36, 51] to handle the indirect calls, and finally build a complete call graph of the Linux kernel. Based on this call graph, given a vulnerable function that includes a DiEH bug, we traverse every entry-point function and extract the shortest path from each of them to the vulnerable function. If there is no path between a vulnerable function to all the entry points, we will mark the bug as non-reachable.

Table 2.5 shows the results of our triggerability analysis. 85.0% of DiEH bugs identified by HERO can be reached from at least one of the entry points, which means that it is possible for adversaries to intentionally trigger these bugs by constructing a specific input. Among these cases, 76.9% of them can be triggered through system calls, which means that they are relatively easier to be triggered by attackers and thus have a higher impact. The last column in Table A.2 shows the specific triggerability information for each bug.

2.6 Discussion

Flow-sensitive vs. Path-sensitive. HERO is flow-sensitive and partially path-sensitive. Being path-sensitive can significantly improve the precision in both pairing and bug detection. However, full path-sensitive analysis cannot scale to large programs such as OS kernel yet. To eliminate the infeasible paths, §2.3.4 showed that HERO employed the symbolic summary to scalably identify conflicting path conditions, and further remove infeasible paths.

Generality. In the evaluation, we applied HERO to both kernels and a userspace program. The evaluation shows that applying HERO to a new program does not require

extra manual effort. However, the precision of pairing analysis and DiEH detection slightly varies on different programs. In general, the detection precision for the Linux kernel is better than it for the FreeBSD and the OpenSSL library. We believe this is due to the reason that the error codes in the Linux kernel are well defined and used. Thus, HERO can better identify error paths and build the EHG.

HERO can be potentially extended also to analyze programs written in other languages or using other error-handling mechanisms. HERO detects DiEH bugs based on two factors (1) capturing errors and (2) analyzing the error-handling code. The logic of developers performing cleanups in error handling is mainly independent of the languages. However, factor (1) is dependent on the languages. To extend HERO, we need to instruct it to identify the errors and error-capturing mechanisms dependent on languages. For example, C++ typically uses the "try-catch" blocks, so HERO needs to further recognize the corresponding patterns in LLVM IR.

Exploitability of detected bugs. To further explore the security impacts of identified DiEH bugs, we need to determine the exploitability of these bugs. However, in this paper, we focus on detecting DiEH bugs instead of exploiting them. We believe that bug exploitation is a separate research topic and is out of our scope. To exploit DiEH bugs, the key is to trigger the corresponding errors, so that the error paths can be executed, which has been demonstrated by the previous works such as fault injection [19] and memory exhaustion [18]. Memory leak and refcount leak bugs can already cause the DoS problem if they can be steadily triggered through these techniques. For other DiEH bugs, after being triggered, adversaries can reuse existing attack techniques such as memory collision attacks [52] to generate the exploits.

Suggestions for avoiding DiEH. Based on our interactions with the kernel maintainers, we suggest several ways to avoid DiEH bugs. First, program developers should try to separate the cleanup operations from normal executions and handle the errors uniformly with a standardized error-handling structure. As shown in Figure 2.1, all the cleanup functions are called after the jump target unreg_video_dev.In contrast, in some cases, only parts of follower functions are used with a standardized error-handling approach, like this example, but other follower functions are called directly after the errors. This inconsistent error-handling often makes the code hard to maintain and can further lead to DiEH bugs. Second, API developers should follow the programming convention

and provide clear instructions. For example, [37] shows the source code of function kobject_init_and_add(). In the latest version of the kernel, the comments clearly emphasize that "If this function returns an error, kobject_put() must be called to properly clean up the memory associated with the object," which, however, is missed before v5.2 and further incur lots of API misuse errors. This information can guide API users to correctly use this API. Third, API users should read instructions to understand how to use the API, instead of assuming its usage. At last, API users can cross-check the usages of API by looking into how other caller functions use the API. Fourth, checking the related patches of this API (e.g., through git log) is also helpful to know the common mistakes.

More applications of pairing analysis. Pairing analysis can be used in other areas, such as helping API users check function usage and bug detectors identify other types of bugs. For example, by identifying the lock/unlock function pairs, we can infer the functions that can execute concurrently and further detecting potential race conditions. These function pairs can be used to detect temporal bugs based on different temporal rules.

2.7 Related work

Function pairs detection. As we compared in §2.5.2, several previous works also try to identify function pairs in large programs. In particular, Mao et al. [12] focused on identifying refcount-related bugs by comparing the inconsistent paths. To this end, they collected 800 pairs of refcount-related APIs by simply string-matching function names, e.g., *_inc and *_dec. WYSIWIB [40] analyzes the data dependencies of pointers to collect 304 pairs of allocation and deallocation functions. Compared to these works, HERO is not limited to a specific type of pair, and its delta-based pairing is more precise. PF-Miner [1] and PairMiner [2] have been introduced in §2.5.2, which employ data mining and string matching. To the best of our knowledge, PairMiner represents the state-of-the-art in automatically detecting various types of function pairs. Compared to HERO, since PF-Miner and PairMiner employ simple mining to collect pairs, we believe that the tools cannot support custom functions and are likely to suffer from precision issues, although they do not evaluate precision. Different from these static analysis tools,

Bai et al. [53] employed dynamic tracing to collect 81 function pairs in four device drivers in Linux, which is not representative of the whole kernel.

Error-handling analysis. Many previous works also analyze error-handling code to detect bugs in software like OpenSSL and OS kernels. Rubio-González et al. [20] and EIO [13] detect error-propagation bugs in file systems. APEx [21], ErrDoc [22], and EPEx [23] reason about the error-code propagation in open-source SSL implementations, either automatically or via user definitions. Saha et al. [54] proposed an automatic approach, which can transform the coding style and structure of the error-handling code to a goto-based standardized error-handling strategy. Tang [55] proposed a tool to detect error code misuses in system programs. EESI [56] is a static analysis tool, which can infer C program function-error specifications through return-code idiom. EESI can identify inadequate and inverted error-checks, and also incomplete error handling bugs. An inherent difference is that these works focus on reasoning about the "handling" itself—if an error code is returned, passed, or handled in callers—instead of the cleanup operations before the handling.

Unlike previous works that aim to make error handling sufficient, EeCatch [57] instead detects exaggerated (or excessive) error handling which often causes crashes. EeCatch employs spatial and temporal cross-checking to identify irregular and *over-severe* error handling as potential exaggerated error-handling bugs. HERO differs from EeCatch in both research goals and approaches. First, HERO aims to detect the ordering issues in the error-handling code, instead of the incorrect severity level of error handling. DiEH causes not only crashes but also memory corruption. Second, HERO's key technique is the precise function pairing while EeCatch features the spatial and temporal cross-checking. To explore the structure of error-handling code, Thummalapenta et al. [58] proposed a mining algorithm, which mining sequence association rules and rule violations of function calls in a large number of the normal and error paths. Different from this work, HERO can precisely identify function pairs based on delta analysis, which can handle the custom functions.

Bug detection in error paths. There is also a line of research that focuses on finding bugs in cleanup operations in error paths. In particular, Saha et al. [11] proposed Hector, which identifies missing resource-release functions in the systems software. Hector

assumes the pointer-returning functions are allocation functions, and the last pointer-usage function is a deallocation function. They identify the missing-release bugs by comparing the inconsistencies in different error paths. Mao et al. [12] implemented RID, which can identify refcount related bugs by analyzing the inconsistent paths in the function; oftentimes, the bugs are in error paths. Lawall et al. [59] proposed a tool to detect error-handling bugs in the Linux kernel and OpenSSL, which are related to API usage protocols. GUEB [60] and CRED [61] are static-analysis tools that can identify use-after-free bugs. All these works focus on a specific type of error-handling bugs, such as missing release. To the best of our knowledge, none of the tools could detect incorrect-order and redundant DiEH bugs, which requires precise and comprehensive identification of function pairs.

Bug detection with rules inference. Some previous works also identified bugs though rules inference based on code semantics. APISan [62] detects API misuses by analyzing rich symbolic contexts. Acharya et al. [63] proposed a mining technique to check the partial-order rules of API usages and detect related rules violation bugs. Gruska et al. [64] presented a tool to mine API usage rules across different projects. Similarly, some previous works [65, 66, 67, 68, 69] detect different types of bugs in a program through a mining approach to generate rules and detect violations. Different from these works, HERO does not rely on unknown-rule mining to detect bugs, thus it can support custom functions; instead, HERO takes advantage of the unique structures of the error-handling code.

Chapter 3

SID: Mining Security-related Functionality Semantics for Precisely Characterizing Security Impact Patches

Major system programs receive an overwhelming number of bug reports, and dealing with these bug reports is much of the life-cycle cost of the software. For instance, Mozilla developers dealt with almost 300 bugs per day in 2005 [70], and a similar rate of new bugs are received in the Mozilla bug database [71] today. The Linux kernel also experiences this problem. As of August 2019, more than 855K patches have been applied by kernel maintainers [72], and the actual number of submissions examined is even higher because many proposed patches are not applied, or require several rounds of revision. Linux also receives many proposed patches from external contributors. Sometimes, these patches fix important bugs, while other patches fix general bugs or even insignificant bugs. Therefore, maintainers must manually review and prioritize the submitted patches to decide if they should be applied immediately or not. Large-scale commercial software development faces similar challenges with bug reports from internal testers and changes proposed by less-experienced developers. This work is time-consuming and error-prone. For example, Hooimeijer et al. [73] showed that 70% of the total life-cycle cost of software is consumed

by maintenance, such as modifying existing code and dealing with bugs.

Given their limited resources, maintainers have to prioritize which bugs to fix by assigning bugs to different priority levels. Highest-priority bugs, such as an obvious security vulnerability, must be fixed immediately. However, lower-priority bugs may be fixed slowly, remain unpatched for a long period of time, or fall through the cracks completely. The common practice is for maintainers to assess the security impacts of bugs manually [74], which is not only challenging and expensive, but also error-prone. This manual classification requires considerable human effort and requires code maintainers to have wide security domain knowledge.

If the security impacts of a critical bug cannot be correctly identified, it will be treated as a lower-priority bug, which will lead to serious security problems. For instance, Arnold et al. [75] described a high-impact compromise of servers for Debian Linux made possible by a Linux kernel vulnerability for which a patch had been available for eight weeks. The Debian administrators had not updated the kernels because the security implications of the patch were not clear until after it was used in a successful exploit. Arnold et al. called this kind of bug a hidden-impact vulnerability: one that is not identified as a vulnerability until after it is made public and potentially exploited by attackers. Their work shows that 32% of vulnerabilities in the Linux kernel were hidden impact vulnerabilities before they were publicized.

Lack of reliable information about the security impacts of bugs is even more critical when open-source software is used in other projects. For example, the Linux kernel is widely used and customized by a large number of platforms such as the Internet of Things (IoT) devices and mobile devices (most prominently Android). A 2018 survey reported by Hall [76] shows that more than 70% of IoT developers use Linux, and Android had a 75% share of the worldwide mobile OS market as of April 2019 [77]. Given the fragmentation of versions and uses of the Linux kernel, it would be impossible for every patch to be promptly applied to every Linux-based device. Instead, patches must be prioritized based on their severity. For instance, under the Android Security Rewards Program [78], reporters are typically required to demonstrate the reproducibility and impact of the reported bugs; otherwise, the reported patches will likely be declined. Previously, we reported three new NULL-pointer dereference bugs to the Android Security team without mentioning their security impacts or providing a proof-of-concept exploit. We considered

these to be vulnerabilities because they can cause DoS, but the Android Security team declined the patches because we did not prove the security impact of the bugs. The empirical results we report also show that bugs that cannot be determined to have security impacts may not be fixed promptly, and thus may introduce serious security problems.

Given the significance of security bugs, many recent papers [79, 80, 81, 82, 83, 84, 85] have attempted to distinguish security bugs from general bugs automatically. Most of these works focus on analyzing textual information, such as a bug description, and their classification of security impacts is mainly based on text-mining techniques. A fundamental limitation is that such classification is highly dependent on the quality of the textual information, which in turn depends on the experience and security-related knowledge of the reporters. Unfortunately, our results indicate that, in many cases, the reporters themselves are not aware of the potential security impacts of the bugs they report. We found that 60.8% of vulnerability patches for the Linux kernel do not mention security impacts in the patch description or subjects. Thus, we cannot expect any classification based on this textual information to reliably classify security bugs. This observation is consistent with recent results by Chaparro et al. [86] which show that many textual bug reports are missing important details and the measurements of Tian et al. [87] which suggest that bug severity ratings are unreliable (i.e., they differ even for duplicate reports of the same bug). To identify security-related patches precisely, we need a more reliable approach to determine the security impacts of bug patches based on code instead of prose.

Existing automated tools also do not provide sufficient support to analyze the security impacts of bugs. Static-analysis tools can warn about code constructs that may have security impacts when misused. However, they generally do not analyze all the factors that affect security impacts. Instead, they make conservative assumptions and thus produce a significant number of false-positive reports that must be filtered out in another step. Providing a proof-of-concept exploit ("PoC") is strong evidence of security impacts, but it requires every patch to include an exploit, which would be a major burden on bug reporters. Bug-finding tools based on fuzzing [88, 89] or whole-program symbolic execution [90, 91] often create a PoC when detecting a problem, but such tools currently generate only a minority of kernel-bug reports, because of challenges such as

state explosion and modeling hardware devices. We would like the process of fixing a vulnerability to be faster than the process of exploiting it if defenders are to stay ahead of attackers. Thus we need an approach to assess the security impact of a bug that is easier than generating a PoC. For adoption, such a tool must have a low false-positive rate and the ability to relate the results to the specific security impact that is implicated. An analysis tool must be trustworthy to convince developers to take a second look at a patch they would otherwise pass by.

In this paper, we propose an automated system, SID, to determine the security impacts of bugs, given their patches. Using security rules that capture common security impacts, SID distinguishes unsafe (rule-violating) and safe (rule-compliant) behaviors of patched and unpatched code, which allows SID to characterize the security impacts of a bug. SID employs differential, under-constrained symbolic execution to match a security risk that is fixed by a patch. The intuition is that both security rules and the program behaviors that are feasible in the unpatched and patched versions can be captured precisely with symbolic constraints. By comparing security constraints with code, SID can reliably determine: (1) if the unpatched code *must* violate a security rule—the unpatched code has a security problem, and (2) whether the patched code can *never* violate the same security rule—the patched code has eliminated the security problem present in the unpatched code. If both conditions are evaluated to be true, this is a strong confirmation that the patch will fix a security violation, and thus that the bug will have a security impact. More importantly, the conservativeness of under-constrained symbolic execution ensures the reliability and the scalability of SID's determination of security impacts. We use slicing and *under-constrainted* symbolic execution to precisely analyze just the code region that is directly relevant to a patch, making conservative assumptions about interactions with other kernel states. This approach avoids most false positives but without expanding the analysis to the whole kernel or requiring an effort that is equivalent to fuzzing or exploit generation. SID determines security impacts based on the code semantics instead of textual information. Based on the semantics, SID can detect the security bugs reliably and provide details about how a bug can be exploited to cause security impacts. This supports developers in formulating an appropriate response to a security bug.

Our priority is for SID's reports of security impact to be reliable, i.e., with high

precision and few false positives. To achieve this, we are willing to accept false-negative cases where there is a security impact that the current implementation of SID is unable to recognize. Some causes include unusual types of security impacts that are not captured by SID's current security rules and the conservative strategy of under-constrained symbolic execution that may miss some cases. An empirical analysis of SID's false-negative results for known vulnerabilities appears in §3.5.2. Further development to reduce false negatives would expand the benefits of SID, but since the current state of practice does not use automated tools to analyze impact at all, we believe that the best path to adoption and security benefit is to begin with tools whose results developers can easily trust when they signal a security bug.

We have implemented SID based on LLVM as multiple static analysis passes. One is a data-flow analysis pass, which identifies vulnerable operations and security operations; the other is an under-constrained symbolic execution pass, which precisely reasons about security impacts. We choose the Linux kernel as our experimental target because it is one of the most widely used and actively-maintained open-source system programs. The security of the Linux kernel is also important to many IoT and mobile devices. For evaluation, we selected 66K recent git commits from the Linux kernel. From these commits, we identified 54K valid commit patches and finally compiled and analyzed 110K LLVM IR files in total. By analyzing these files, SID successfully found 227 security bugs with a 97% precision rate. These security bugs may introduce security impacts such as out-of-bound access, use-after-free, double-free, uninitialized use, and permission bypass. More critically, we found that 21 of these security bugs are still not patched in Android, which can cause severe security problems for billions of Android devices.

To further confirm that the identified security bugs are vulnerabilities, we analyzed the reachability of the security bugs from attacker-controllable entry points (e.g., system calls) and also reported them to CVE maintainers. As a result, we find that 67.8% of identified security bugs are potentially reachable from entry points. On the other hand, we in total reported 154 security bugs to CVE maintainers and have received 37 responses with 24 new CVEs assigned. The evaluation results show that SID is effective and precise in automatically determining the security impacts of a large number of bug patches.

The rest of this paper is organized as follows. We review background concepts in §3.1, and give an overview of our approach in §3.2. We then present the design of SID in

section §3.3; the implementation of SID in section §3.4; the evaluation of SID in section §3.5; limitations and future work in section §3.6; related work in §3.7.

3.1 Summarizing the Common Security Bugs and Corresponding Functionality Semantics

To propose an effective approach to understand the causes and security impacts of bugs and thus to find security bugs, we analyzed some existing patches for vulnerabilities in the Linux kernel. Specifically, we first show differences between general bugs and vulnerabilities. Then, we analyze the common causes and security impacts of vulnerabilities. Based on the statistical results, we summarize the model and components of the vulnerability patches. After that, we define the problem scope and the assumptions of this work.

3.1.1 General Bugs, Security Bugs, and Vulnerabilities

A bug is a vulnerability if it causes security impacts when triggered. A vulnerability is also called a security-critical bug (or just a security bug), and is distinguished from a general bug. Different kinds of vulnerabilities often differ in security impacts. The example in Figure 3.1 shows the difference between a general bug and a vulnerability. In this example, missing the check in line 3 is a vulnerability because it leads to an out-of-bound access in line 9. In comparison, missing the check in line 6 will not introduce any security impact; thus, it is just a general bug. More details about the definition and detection for security checks can be found in previous work [92].

3.1.2 Common Security Bugs and Impacts

In this work, we aim to cover the most common security bugs and their corresponding security impacts. To this end, we first examined recent Linux-kernel vulnerabilities included in the national vulnerability database (NVD). There are nearly 800 vulnerabilities reported in the past three years, but only a small part of them include valid git-commit information of their patches. Thus, we chose to analyze 100 of them across these years.

Table 3.1 presents the analysis results. The most common causes for security bugs in

```
int Bug_Vuln(unsigned int Type)  {
    char colors[4] = {'r','g','b','-'};
    if (Type > 3)
        return -1;

    if (Type == 3)
        return 0;

    printf("Color Type: %c", colors[Type]);
    return 0;
}
```

Figure 3.1: Differences between a vulnerability and a general bug. Missing the check in line 3 results in a vulnerability while missing the check in line 6 is a general semantic bug.

the Linux kernel are missing or wrong security checks (bound check, permission check, NULL check, etc.), missing initialization, missing or incorrect locks/unlocks, API misuse, and missing nullification. The following results also show the relationship between the root causes of security bugs and their security impacts: (1) missing/wrong bound check typically leads to out-of-bound access; (2) missing initialization often leads to uninitialized use; (3) missing permission check leads to permission bypass; (4) missing NULL check commonly leads to NULL-pointer dereference; and (5) missing pointer nullification leads to use-after-free and double-free.

In the study, we differentiate the bugs (i.e., the root causes) from their security impacts, which are often mixed up in traditional vulnerability classification (e.g., in NVD). While traditional vulnerability classification tends to focus on security impacts, they are not the root causes. For example, missing a bound check is the bug; however, the out-of-bound access caused by the missing-check bug is the security impact. As such, bug patches typically fix the root causes and only indirectly prevent security impacts. Therefore, to determine security impacts, we need to analyze the "effects" of patches. Moreover, we define security impacts based on the security-rule violating operations (e.g., out-of-bound access and use-after-free) instead of the resulting exploits such as information leaks or control-flow hijacking. This is consistent with the goal of SID—determining how a bug results in security-rule violating operations. How these operations can be exploited for an attack is out of our scope.

Common security bugs (root cause)	**Percent of bugs**	**Main security impacts**
Missing/wrong bound check	21%	Out-of-bound access
Missing initialization	9%	Uninitialized use
Missing permission check	9%	Permission bypass
Missing NULL check	7%	NULL-pointer dereference
Missing/wrong locks/unlocks	6%	Use-after-free, double-free, Permission bypass, NULL-pointer dereference
API misuse	5%	Out-of-bound access, Permission bypass, Uninitialized use
Missing error-code check	5%	Out-of-bound access, Uninitialized use, NULL-Pointer dereference
Missing pointer nullification	4%	Use-after-free, double-free
Others such as numerical errors	34%	Uninitialized use, Out-of-bound access, NULL-pointer dereference, Others

Table 3.1: Common security bugs and security impacts.

3.1.3 Patch Model and Components

To determine the security impacts with a given patch, we first need to identify the components in the patch that are related to security impacts and to build a patch model. Based on our empirical analysis of existing patches for vulnerabilities, we identify three key components in determining security impacts. We then create a patch model that incorporates the components, as shown in Table 3.2. In this model, the three components are security operations, critical variables, and vulnerable operations. (The symbol, +, indicates security operations introduced by the patch).

- **Security operations** are used in patches to fix at least one security impact. Table 3.1 shows that missing or wrong security operations are the most common root causes for security bugs. From those statistical results, we summarize the common security operations: security checks (e.g., bound checks, permission checks), initialization operations, lock or unlock operations, and pointer nullification.
- **Critical variables** are the ones whose (invalid) values or status can lead to security impacts. As such, critical variables are typically targeted by security operations. For example, a checked bound variable is a critical variable.

- **Vulnerable operations** signal the risk of a security bug, often because they can behave unsafely. Based on our study, common vulnerable operations include buffer and array operations, read or write operations, pointer operations, operations involving critical data structures such as inodes or files, and resource-release operations.

1.+ Security_op(CV, ...)
...
2. Vulnerable_op(CV, ...)

Table 3.2: The common patch model and the three key components: security operation, critical variable (CV), and vulnerable operation. The security operation is typically added or updated by a patch.

This model shows that patch updates or adds new security operations in the vulnerable code to eliminate the security impacts which are introduced by the vulnerable operations. Most commonly, a security operation is inserted before a vulnerable operation to prevent an unsafe state. Evaluation results in §3.5.5 show that about 88% of vulnerabilities in the Linux kernel can precisely or partially fit into this model. Thus, we can use this model to determine security impacts for most patches.

3.1.4 Problem Scope and Assumptions

In this work, we analyze the bug patches in the Linux kernel. We choose the Linux kernel as the target program for the following reasons. (1) The Linux kernel is a foundational and widely used program. Many other operating systems are based on the Linux kernel, such as Android. Security bugs in the Linux kernel may introduce critical security impacts in all Linux-based systems. Thus, applying patches for security bugs in the Linux kernel is vital. (2) The Linux kernel is an open-source program with a well-maintained patch history, which facilitates patch-based analyses. These reasons motivate us to choose the Linux kernel as the experiment target. However, SID is general and applicable to other similar software such as FreeBSD and Firefox, as discussed in §3.6.

We assume that the provided patches correctly fix actual bugs. We may not correctly obtain the security impacts of vulnerabilities if the patches are incorrect. Based on the statistical results in Table 3.1, we choose to determine all the common security impacts listed in Table 3.3. The current version of SID does not include NULL-pointer dereference

because it is difficult to exploit in the Linux kernel—the zero page is protected against being allocated. However, NULL-pointer dereference can be naturally supported by modeling the "non-NULL" as a constraint and the NULL dereference as a vulnerable operation. Table A.5 shows how to use our model to cover the patches for other common types of vulnerabilities. Also, more details about extending SID to detect more types of bugs are discussed in §3.6.

Common security impacts (%)	**Common security operations (%)**
Permission bypass (21.9%)	Permission check (59%) Changing permission flags (8%) Others (33%)
Out-of-bound access (16.5%)	Bound check (79%) Reset the size of buffer (10%) Others (11%)
Uninitialized use (13.7%)	Initialize the variable (78)% Others (22%)
Use-after-free/double-free (4.3%)	Pointer nullification (32)% Lock or unlock operations (25%) Others (43%)

Table 3.3: Common security operations for fixing common security impacts.

To determine common security operations related to these impacts, we selected 100 recent vulnerabilities for each security impact in the Linux kernel from NVD. By manually checking the patches of these vulnerabilities, we count the security operations as shown in Table 3.3. Based on this result, we choose to cover the most common security operations for each security impact including: (1) bound checks, (2) initialization operations, (3) permission checks, and (4) pointer nullification. In our current implementation, we do not include other security operations such as lock or unlock operations because either they are not common or they predominantly cause non-security impacts such as incorrect results. In total, based on Table 3.3, by calculating the proportion of these covered security operations against these common security impacts, the current implementation of SID can support about 38% of vulnerabilities. However, SID's approach is generic, and covering more types of vulnerabilities requires only extra engineering efforts for modeling and identifying the three patch components. Table A.5 shows how to support several more types of bugs which can cover 13% more of vulnerabilities. More discussion can be found in §3.6.

We further assume that the bug fixed by a patch is triggerable, which means that, by

providing specific inputs, the execution can reach the buggy code. Existing techniques, such as guided fuzzing [93, 94] and symbolic execution over untrusted inputs [90, 91], can search for inputs that trigger a bug. However, determining how to trigger a bug can be challenging; if a bug has security impacts, it is usually worthwhile to fix it, even if it is not obvious to be triggerable.

3.1.5 Security Rules

Security impacts occur when security rules are violated. To precisely determine if a security impact exists, we also need to define the corresponding security rule. The specific security rules help SID construct constraints that can be solved. With security rules, the determination of security impacts can be transformed into a constraint-solving problem. For the security impacts shown in Table 3.3, we define the corresponding security rules as follows, which are consistent with the standard definition in CWE [43].

- **Out-of-bound access.** Memory read and write operations should be within the boundary of the current object.
- **Use-after-free and double-free.** An object pointer should not be used after the object has been freed.
- **Uninitialized use.** A variable should not be used until it has been initialized.
- **Permission bypass.** Permissions should be checked before performing sensitive operations such as I/O.

3.2 Overview of SID

Given a patch and the target program, SID automatically determines if the patch fixes some security impacts. In this section, we show the approach and workflow of SID.

3.2.1 The Approach of SID

We now use an example, shown in Figure 3.3, to illustrate the *symbolic rule comparison* approach of SID. This is an out-of-bound access security bug that is fixed by a patch that inserts a bound check in line 6. Given this patch, the goal of symbolic rule comparison is

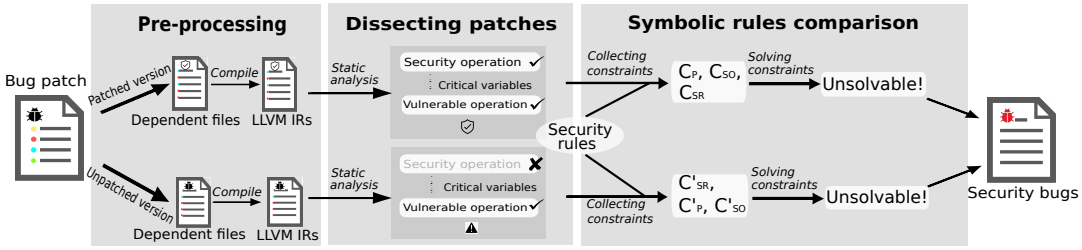

Figure 3.2: Overview of SID. C_SO = Constraints from security operations, C_SR = Constraints from security rules, C_P = Constraints from paths.

to confirm whether the patch fixes violations of some security rules, e.g., out-of-bound access, that introduce security impacts.

Symbolically analyzing patched code. First, SID analyzes the patched version to *prove that it will never violate a security rule.* In function iwl_sta_ucode_activate, SID identifies the security operation in the patch, bound check in line 6. Then, SID extracts the critical variable, sta_id, from the security operation. By using data-flow analysis, SID identifies six potential vulnerable operations located in lines 11, 14, 16, 19, 21, and 23. Each pair of security operation and vulnerable operation defines a slice, and SID performs under-constrained symbolic execution against each slice. SID will construct and collect three sets of constraints. The first set of constraints are constructed from the security operation of the patch, e.g., sta_id $<$ IWLAGN_STATION_COUNT, in the example. The second set of constraints is collected from the slice through symbolic execution (e.g., capturing if a variable is modified via arithmetic). After that, SID *artificially* constructs the third set of constraints that represent the *violation* of a security rule, e.g., memory access out of the bound of stations. All the three constraint sets are then *merged* as the final constraint set. Finally, SID employs a constraint solver to prove that the final constraint set is *unsolvable*, which means that the slice will not violate the security rule.

Symbolically analyzing unpatched code. Second, if the patched version never violates a security rule, SID analyzes the unpatched version to *prove that every behavior removed by the patch would violate the security rule.* A slice in the patched code is matched with one in the unpatched code based on its critical variables and vulnerable operation, as described in more detail in §3.4.4. Similar to the analysis for the patched code, SID constructs and collects three sets of constraints for the unpatched code, however, in a different way. Specifically, the first set corresponds to the security operation in

```
/* Linux: drivers/net/wireless/intel/iwlwifi/dvm/sta.c
 * CVE-2012-6712 */

int iwl_sta_ucode_activate(... , u8 sta_id)
{
+   if (sta_id >= IWLAGN_STATION_COUNT) {
+       IWL_ERR(priv, "invalid sta_id %u", sta_id);
+       return -EINVAL;
+   }

    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
            "addr %pM\n",
            sta_id, priv->stations[sta_id].sta.sta.addr);

    if (priv->stations[sta_id].used) {
        IWL_DEBUG_ASSOC(priv,
                "STA id %u addr %pM already present in uCode"
                sta_id, priv->stations[sta_id].sta.sta.addr);
    } else {
        priv->stations[sta_id].used |= IWL_STA_UCODE_ACTIVE;
        IWL_DEBUG_ASSOC(priv, "Added STA id %u addr %pM\n",
                sta_id, priv->stations[sta_id].sta.sta.addr);
    }

    return 0;
}
```

Figure 3.3: A missing bound-check bug and its patch (lines 6-9).

the patch. Since the security operation is not in the unpatched code, SID will add an artificial constraint that is the *opposite* of the constraint for the security operation; it is important to note that this captures the behaviors that are blocked by the security operation in the patched code. For example, while the security operation is to ensure sta_id < IWLAGN_STATION_COUNT, SID will instead add a constraint, sta_id >= IWLAGN_STATION_COUNT. The second constraint set is similarly collected from the slice through symbolic execution. The last constraint set corresponds to the security rule. However, the constraints are constructed in such a way that they satisfy rather than violate the security rule, e.g., in-bound access of stations. Finally, the three constraint sets are merged and proved to be *unsolvable*, i.e., the unpatched code will violate the security rule in all the cases blocked by the patch. The first and the third constraint sets of the unpatched version are constructed in a opposite way as in the patched version, which allows SID to leverage the conservative *unsolvability* of under-constrained symbolic

execution to precisely determine the security impacts.

Confirming security impact. At last, if both cases are indeed unsolvable, SID confirms that the patch is a fix for a security-rule violation. In other words, the corresponding bug is a security bug. Symbolic execution allows SID to precisely check the match between a patch and the corresponding security rule, while the under-constrained approach makes the analysis conservative about parts of the code outside the analysis scope.

3.2.2 The Workflow of SID

Figure 3.2 is an overview of SID including the following three main phases: pre-processing, dissecting patches, and symbolic rule comparison.

Pre-processing. The pre-processing phase includes two tasks. First, given a specific git commit, it ensures that it is a bug-fixing patch. Commits for branch merging, documentation, and code formatting are eliminated. Second, it identifies the dependent files of the patch, which are collected using standard control-dependency analysis against the patched code and taint analysis against the variables involved in the patch, in both a forward and backward manner. We found that the dependent files are a single file containing the patch in most cases. It then prepares the patched code and unpatched code (by reverting the patch) and invokes LLVM to compile them into LLVM IR files.

Dissecting patches. In this phase, SID dissects the patch to identify the key components according to the patch model (§3.1.3). SID first identifies the security operations in both the patched and unpatched versions. SID then extracts involved critical variables from the identified security operations. Next, SID applies data-flow analysis against critical variables to collect vulnerable operations. The data-flow analysis also helps SID collect slices from security operations to vulnerable operations for the critical variables. It is worth noting that security operations and vulnerable operations have a many-to-many mapping, which means that a single patch may have multiple slices. After obtaining these slices, SID further employs symbolic execution to test the feasibility of the slices by testing if the path conditions of the slices are satisfiable. If the conditions are unsatisfiable, the path is infeasible, and is immediately discarded.

Symbolic rule comparison. For the feasible slices, SID then performs a symbolic rule comparison to confirm the security impacts of the bug with the approach described

in §3.2.1.

3.3 Design of SID

In this section, we present the design of SID. In particular, we focus on the static analysis for dissecting patches and symbolic rule comparison for determining security impacts.

3.3.1 Static Analysis for Dissecting Patches

Given a patch and the target program, SID employs static analysis to (1) identify security operations, critical variables, and vulnerable operations and (2) construct slices from security operations to vulnerable operations for the critical variables. According the the patch model (§3.1.3), we summarize the patch patterns for each class of vulnerability in Table 3.4.

Out-of-bound access	**Permission bypass**
1.+ Security_ck(Bound); ... ... 2. Vulnerable_op(Bound, ...);	1.+ ret = Perm_func(CV, ...); 2.+ Security_check(ret); ... 3. Vulnerable_op(CV, ...);
Use-after-free or double-free	**Uninitialized use**
1. free(Pointer); 2.+ Pointer = NULL; ... 3. Vulnerable_op(Pointer, ...);	1.+ Initialize(CV); ... ... 2. Vulnerable_op(CV,...)

Table 3.4: Patch patterns for different classes of vulnerabilities and the key components in the patches. + denotes the security operation in the patches, and Vulnerable_op represents some vulnerable operations.

Identifying security operations. First, SID analyzes the patches to identify security operations. As described in §3.1.4, SID identifies four kinds of security operations: permission check, bound check, initialization operation, and pointer nullification. Based on the statistical results in Table 3.3, we summarize common patterns of patches in Table 3.4. These patterns describe how security operations fix the corresponding security impacts caused by the vulnerable operations. For out-of-bound access vulnerabilities, the patches typically insert the security operation, bound check, to make sure in-bound access against a memory object. For permission bypass vulnerabilities, the patches insert

permission check as the security operation. The permission checks are usually done by checking the return value of permission functions. For use-after-free and double-free vulnerabilities, the patches often insert pointer nullification as the security operation. Since, in the Linux kernel, NULL check is typically enforced before using a pointer, nullification becomes a common way for fixing use-after-free. For uninitialized-use vulnerabilities, the patches instead initialize a memory object before it is being used. §3.4 will present further details on analyzing the code to identify the security operations.

Extracting critical variables. Next, SID extracts the targets of the identified security operations as critical variables. For nullification, initialization, and bound-check cases, critical variables can be easily identified by extracting the involved variables (not constants). However, the critical variables in permission-check cases can be challenging to identify. The method SID uses to identify such critical variables is based on permission functions (e.g., ns_capable()) which include both critical variable and capability numbers as parameters. Non-constant parameters (e.g., objects such as files, inodes, or subjects such as users) used in such permission functions are typically sensitive resources whose accesses require permission checks. Therefore, SID identifies these parameter variables as critical variables.

Slicing to find vulnerable operations. After extracting the critical variables, SID then uses data-flow analysis to find vulnerable operations using extracted critical variables, i.e., slicing to find vulnerable operations. An operation is regarded as a vulnerable operation if it may introduce security impacts via the critical variables. We do the backward or forward data-flow analysis against the critical variables to match the vulnerable operations, according to the corresponding patterns shown in Table 3.4. With that, we also obtain slices for critical variables that involve both security operations and vulnerable operations. Because vulnerable operations and security operations are many-to-many mappings, one vulnerable operation or security operation can be in multiple slices. More details about collecting vulnerable operations for different kinds of vulnerabilities can be found in §3.4.3.

Pruning slices. After extracting these slices, SID removes slices with the following cases. (1) The critical variables are newly introduced in the patch. In this case, the unpatched code will never use them. (2) The vulnerable operations exist only in the patched version but not in the unpatched version, which means that the vulnerable

operations are also newly introduced by the patch, and corresponding security impacts will not exist in the unpatched version.

Removing infeasible slices. Finally, SID removes infeasible slices using symbolic execution. SID performs the under-constrained symbolic execution for each slice to collect path constraints. When reaching the end of the slice, SID tries to solve the constraints. If these constraints are unsolvable, SID will discard this slice. Removing unsolvable slices will reduce false positives and make sure that the *unsolvability* in symbolic rule comparison must be related to security-rule violations (not vacuous), ensuring the effectiveness of SID.

3.3.2 Symbolic Rule Comparison

SID further performs a symbolic rule comparison to determine security impacts and identify security bugs. SID determines that a patch is for a security bug if the patched and unpatched versions satisfy both of the following requirements.

- The patched version *never* violates a security rule.
- The unpatched version *always* violates the security rule in the situations excluded by the patch.

The checking against the *absolute* requirements is possible because SID uses under-constrained symbolic execution, which is conservative [95]. By combining both requirements, it is intuitive to determine that the patch prevents violation of a security rule. To realize the checking against the requirements, SID first constructs and collects the constraint sets from patches, security rules, and the slice paths for the patched and unpatched code separately. If both constraint sets for the patched and unpatched code are unsolvable, SID determines the security impact.

3.3.3 Constructing and Collecting Constraints

SID constructs and collects three sets of constraints for both the patched version and the unpatched version. These constraints come from three sources—security operations from the patch, the code path of each slice, and security rules. We now describe how SID collects or constructs these constraints.

Security operations	Constraints from security operations	
	Patched version	**Unpatched version**
Pointer nullification	$\text{FLAG}_{CV} = 1$	$\text{FLAG}_{CV} = 0$
Initialization	$\text{FLAG}_{CV} = 1$	$\text{FLAG}_{CV} = 0$
Permission check	$\text{FLAG}_{CV} = 1$	$\text{FLAG}_{CV} = 0$
Bound check	CV $<$ UpBound, or CV $>$ LowBound	CV $\geqslant$ UpBound, resp. CV $\leqslant$ LowBound

Table 3.5: Constraints for security operations from patches. Flag_CV: Flag symbol; CV: critical variable; UpBound: checked upper bound; LowBound: checked lower bound.

Constructing constraints from security operations. The constraints from security operations are used to capture the "effects" of them in preventing security impacts. We define the constraints for each class of security operation for the patched and unpatched code, respectively. Table 3.5 shows our rules for constructing constraints for different security operations. Besides out-of-bound access, constraints for other security operations are used to indicate whether the security operations are present. Therefore, we use a binary-flag symbol to represent the constraint. Specifically, FLAG_CV indicates the status of the corresponding critical variables, in terms of the presence of the security operations. FLAG_$CV = 1$ means that the security operation has been enforced against critical variable CV. By contrast, FLAG_$CV = 0$ indicates the absence of the security operation for CV. The constraints for bound-check cases are more complicated. The constraints are used to limit the upper bound and/or the lower bound of a memory object. In addition to indicating the presence of the security operation, we also need to know the specific value range of the value of the critical variables. Thus, we use symbolized critical variable to represent the value.

To check against the requirements for the patched and unpatched code, we must construct these constraints differently (in an opposite way). For permission bypass, use-after-free, double-free, and uninitialized use, SID inserts constraints FLAG_$CV = 1$ for the security operations in the patched version while inserting FLAG_$CV = 0$ in the unpatched version because the security operations are missing. For out-of-bound access vulnerabilities, SID adds the constraints CV $<$ UpBound and/or CV $>$ LowBound on the symbolized critical variable in the patched version. For example, in Figure 3.3, the constraint from the security operation in the patch is sta_id $<$ IWLAGN_STATION_COUNT. The values of UpBound and LowBound are determined

based on the specific bound-check security operations. In the unpatched version, SID instead inserts the constraints, CV >= UpBound or CV <= LowBound, which are opposite to the ones in the patched version. This is to prove that, without the security operation, an out-of-bound access problem will occur in the unpatched version. For the example in Figure 3.3, SID will insert the constraint sta_id >= IWLAGN_STATION_COUNT for the unpatched version.

Collecting constraints from slice paths. The constraints from paths (from a security operation to a vulnerable operation) are collected from two parts. The first part is the same as the one in removing infeasible slices (see §3.3.1). These constraints are collected from path conditions that are checked to make sure the slice itself is feasible. The second part is to collect manipulations against critical variables. For example, in the uninitialized use case, for an initialization against the critical variable in the slice path, SID will add a constraint, FLAG_CV = 1.

Constructing constraints from security rules. The last set of constraints SID constructs are from the security rules. These constraints are important to evaluate if a security-rule violation may occur. We develop multiple rules for constructing these constraints of different security rules, as shown in Table 3.6. SID also constructs *opposite* constraints for the patched and unpatched versions.

For the *patched* version, we want to prove that, with the protection of the security operations, it is impossible to violate the security rules. Therefore, SID inserts *rule-violating* constraints and hopes that they are unsolvable, i.e., Flag_CV = 0, CV >= MAX and/or CV <= MIN. SID will first employ static analysis to figure out the size of the memory object in use. In the example in Figure 3.3, SID can easily find that the buffer, stations[], is on the stack with a fixed length 16. For other cases, e.g., the buffer is on the heap, SID will use backward data-flow analysis to find the allocation site to determine its size. If the size is a constant, the value will be used; otherwise, for a variable, SID instead symbolizes it. After knowing the buffer size, SID then inserts an *out-bound* constraint (e.g., sta_id >= 16) for the patched version. For the *unpatched* version, we want to prove that, without the security operations, it always violates the security rules. In order to achieve this, the constraints in the unpatched version will be opposite to the constraints in the patched version, which instead represents the compliance of the security rules. Here, we hope to prove that the constraints are unsolvable, so the compliance of

security rules is impossible. For example, in Figure 3.3, from the security rule of in-bound access, SID inserts the constraint, sta_id $<$ 16, for the unpatched version.

Security rules	**Patched version**	**Unpatched version**
No use after free	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
Use after initialization	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
Permission check before sensitive operations	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
In-bound access	CV $\geqslant$ MAX, and/or CV $\leqslant$ MIN	CV $<$ MAX, resp. CV $>$ MIN

Table 3.6: Rules for constructing constraints from security rules. MAX: maximum bound of the buffer; MIN: minimum bound of the buffer.

3.3.4 Solvability for each slice

To know the solvability of each slice, SID merges the three sets of constraints as the final ones for the patched and unpatched versions, respectively. SID then uses SMT solver, Z3, to solve the constraints. In the example in Figure 3.3, for the patched version, the final constraint set is sta_id $<$ IWLAGN_STATION_COUNT && sta_id $>=$ 16, which are generated from the security operation and security rules. Similarly, for the unpatched version, the final constraint set is sta_id $>=$ IWLAGN_STATION_COUNT && sta_id $<$ 16. Both final constraint sets for the patched and unpatched versions are unsolvable, because IWLAGN_STATION_COUNT is 16.

3.3.5 Comparison against symbolic rules

Finally, after solving these constraints, SID compares the results of solvability to determine security impacts. A patch is determined to fix a security impact if the constraints for both the patched and unpatched versions are *unsolvable*, which means that the patched version must not violate the security rule, and the unpatched version must violate the security rule. Therefore, the patch fixes the violation against the security rule. Thanks to the conservativeness of under-constrained symbolic execution, the determination is precise. In the example in Figure 3.3, SID finds both constraints in the patched and unpatched versions unsolvable, so this patch fixes an out-of-bound access problem. If either constraint set is solvable, SID disqualifies the bug fixed by the patch as a security bug.

3.4 Implementation

We have implemented SID on top of LLVM with multiple passes for finding security checks, symbolic execution, and data-flow analysis. SID in total contains 5.6K line C++ code and 1.2K line Python code. The rest of this section presents important implementation details of SID, including preparing analysis environment, collecting vulnerable and security operations, matching information between the patched version and the unpatched version, and the symbolic-execution engine.

3.4.1 Preparing Analysis Environment

Since not all of the patches fix bugs, we use script code to eliminate common non-bug commits such as branch merging, documentation, and indentation updating. In total, 17.4% of git commits are classified as non-bug commits and thus eliminated, which improves the efficiency of SID. For the remaining patches, SID continues to generate LLVM IR and dissect them.

Preparing LLVM IR. Since SID is based on LLVM, the patches should be compiled into LLVM IR with a patched version and an unpatched version. Because the corresponding vulnerable operations can be in other source files than the one containing the patch, we employ static analysis to extract the dependent files from the patch automatically. We first extract the patch code and variables involved in the patch. Then, we employ standard control-dependency analysis against the patched code and taint analysis against the involved variables, in a both forward and backward manner, to identify all dependent files. Interestingly, we found that, in most cases, the file containing the patch also contains the vulnerable operations. The unpatched version is obtained simply by checking out the git commit right before the one for the patch. Finally, we invoke clang to compile these files, for both the patched version and unpatched version, into LLVM IR.

To mitigate the path-explosion problem in the under-constrained symbolic execution, we unrolled loops in the LLVM IR level by treating them as if statements, which is a common practice adopted by recent techniques [96, 97]. To identify indirect-call targets, we take recent advances that use struct types to match the function targets [98, 99, 100].

3.4.2 Identifying Security Operations

As shown in §3.3, in dissecting patches, SID first identifies security operations. By analyzing the patch code in the git log, SID can tell if a patch contains security operations. However, the security operations can be intended for new variables introduced in the patch. In this case, the security operations are not aimed at fixing bugs in the unpatched code. To eliminate these cases, SID ensures that the critical variables are also in the unpatched code. Further details are presented in §3.4.4. For different types of security impacts, the corresponding security operations are different. We identify security operations as follows.

- **Bound checks.** We regard bound checks as security operations. We use two criteria to identify bound checks. (1) A bound check uses a conditional statement such as if statement. The operator of the comparison instruction should be =, >, <, >=, or <=, and both of the operands of bound check should be of integer type such as int or unsigned. (2) One branch of the conditional statement should result in error handling (e.g., returning an error code) when a bound check fails while other branches continue normal execution. This is similar to the check definition in [99, 101].
- **Pointer nullification.** NULL checks are typically enforced before using pointers. Based on the common patch patterns, we regard pointer nullification as the security operation against use-after-free. Nullification can be easily identified when NULL is assigned to a pointer.
- **Initialization.** We regard initialization operations as the security operations against uninitialized use. Initialization is either a store instruction that assigns 0 to a variable or a call to memset() that takes 0 as the value argument.
- **Permission checks.** Permission check is the security operation against permission bypass. By looking into how permission bypass is commonly patched, we first empirically collect the common permission functions such as afs_permission() and ns_capable(). Then, we identify a conditional statement (e.g., if statement) as a permission check if it is a security check [99] against the return value of these permission functions.

3.4.3 Identifying vulnerable operation

SID then identifies vulnerable operations. To do that, SID first extracts the critical variables from the identified security operations, as described in §3.3.1. Based on the uses of the critical variables, SID employs data-flow analysis (i.e., slicing) to identify the vulnerable operations, according to the rules in §3.1.3. Here, we have extracted the critical variables from security operations. With that, we present some implementation details on the identification of vulnerable operations that use the critical variables.

- **Out-of-bound access.** We first identify the instructions that access an array or buffer using the critical variables (i.e., size variables) as vulnerable operations. We also identify common read or write functions (e.g., memcpy()) that take as input the critical variables as vulnerable operations.
- **Use-after-free and double-free.** We conservatively identify all pointer dereference operations that target the critical variables as vulnerable operations.
- **Uninitialized use.** We identify the common operations on the uninitialized variables as vulnerable operations. These common operations include pointer dereference, function calls, memory access, and binary operations such as arithmetic operation. Also, these operations must also target the critical variables.
- **Permission bypass.** Based on the extracted the critical variables in permission checks, which take struct types such as kuid_t, inode, file or corresponding pointer types, we conservatively identify operations against critical variables as vulnerable operations.

3.4.4 Mapping Operations in Patched and Unpatched Versions

A patch may involve multiple security operations and vulnerable operations, thus multiple slices. To perform the symbolic rule comparison, we need to map the corresponding slices in the patched and unpatched versions. SID pairs slices relying on various types of information such as function name and control flow. Specifically, to pair the slices, SID first extracts the vulnerable operation of the slices for both the patched and unpatched versions. The vulnerable operations must exactly match. If the vulnerable operations are matched, SID further employs control-flow comparison to make sure that the two

slices are the same except the parts introduced by the patch. With these two steps, SID can map the slices between the patched version and the unpathed version.

3.4.5 The Under-Constrained Symbolic-Execution Engine

SID uses under-constrained symbolic execution to analyze the code for patched and unpatched versions. Similar to UC-KLEE [95], the symbolic execution of SID can start from any point in a function. Specifically, SID only executes on the slices collected from the static analysis during dissecting patches. Since the collected constraints in these individual slices are not complete, they are under-constrained, which may lead to false negatives. However, SID's main goal is to determine security impacts with a low false-positive rate. We will discuss how to collect more constraints beyond the slices in §3.6.

Avoiding path explosion. Path explosion is a general problem in symbolic execution, which is fortunately mitigated in SID. Different from the whole-program symbolic execution, SID only works on the slices collected by the static analysis. Most of the security operations are near the vulnerable operations, so most slices are short, involving a single module. As such, the path-explosion problem, in most cases, does not occur. However, we did observe the path-explosion problem in some instances. To completely avoid path explosion, SID chooses to discard slices with more than 150 basic blocks. This threshold number is carefully selected based on our statistical study—more than 98.8% of slices cover less than 150 basic blocks. The heuristic is also used in previous works, such as UC-KLEE [95], to alleviate path explosion. By using this method, slices can be symbolically analyzed quickly without encountering the path-explosion problem.

3.5 Evaluation

We evaluate the accuracy, effectiveness, and scalability of SID, and also present new findings regarding characteristics of security bugs. We chose the Linux kernel as the target program, and collected more than 66K git commits in recent years. During the pre-processing, 11,433 non-bug commits are eliminated, which finally returns us 54,651 valid patches. For these valid patches, we generated 110,136 LLVM IR bit code files for both the patched and unpatched versions. The experiments were performed on Ubuntu

18.04, 64-bit OS with LLVM-8.0. The machine has a 32GB RAM and is equipped with six cores Intel (R) Xeon (R) W-2133 CPU @ 3.60GHz. It is worth noting that the following measurement uses a single thread without parallel computing.

Efficiency and scalability. For each patch, SID analyzes both the patched version and the unpatched version. The analysis takes an average of 0.415 seconds (median 0.056s, max 15s) for each version, which means that, for every patch, the analysis costs 0.83 seconds on average. During the analysis, the detection of out-of-bound read or write vulnerabilities patches takes 63 % of the total time, while all other cases take only 37% of the total time. Processing patches for out-of-bound access vulnerabilities requires more time since some more slices and constraints that are more complex, requiring more time for symbolic execution. The results indicate that SID is efficient enough to handle a massive amount of patches precisely.

3.5.1 False Positives of SID

We use precision, |TP| / |TP+FP|, to evaluate the false positives of SID, where TP and FP are the numbers of true positives and false positives. To calculate the precision, we manually checked all the results generated by SID. In order to precisely confirm that these bugs are true security bugs, we look into the patch description (comments), the patch code, and the involved source code. If the comments have already mentioned the same security impacts as SID found, we regard them as a security bug because both Linux maintainers and reporters have confirmed the security impact. Otherwise, we manually review the patch code and the involved source code to check (1) if the vulnerable operations found by SID indeed introduce security impacts in the unpatched version and (2) if these security impacts are eliminated by the security operations in the patch. If both of these conditions are true, we confirm the security impacts and the security bug. Finally, we confirmed 227 security bugs with 8 false-positive cases. As a result, the precision rate of SID is 97%. We investigated these false positives and summarized the reasons as follows.

Missing constraints in preventive-patching cases. SID employs under-constrained symbolic execution to analyze only the slices that start from the security operations to the vulnerable operations. As such, earlier constraints that are before the security operations

will be missed. In general, if the "earlier" constraints have already been able to prevent a security impact, the constraints in the patch are unnecessary. However, we did find five cases in which the patches are preventive and enforce redundant constraints. Developers enforce the preventive patches because the "earlier" constraints can be changed in future code. In these cases, SID will identify them as patches fixing security bugs, leading to false positives. Eliminating all these preventive patches is a hard problem, which requires a more complete constraints set. However, we would like to mention that these five cases violate SID's threat model—the provided patches correctly fix actual bugs.

Inaccurate static analysis. During dissecting patches, SID employs static data-flow analysis to find the slices from security operations to vulnerable operations. Due to the inaccuracy of the static analysis and the incompleteness of identifying security or vulnerable operations, many slices are infeasible. Although SID further employs symbolic execution to validate the slices, because the symbolic execution is under-constrained, the resulting slices may still be infeasible, leading to false positives. The remaining three false positives are caused by such inaccuracy. In the future, we plan to improve the under-constrained symbolic execution by collecting more constraints, as discussed in §3.6.

3.5.2 False Negatives of SID

By design, SID aims to ensure less false positives by allowing more false negatives. In this section, we evaluate the false negatives of SID and investigate the causes. Generally, the evaluation of false-negative cases for static analysis tools is not as easy as a precision evaluation because it requires a ground-truth set. To this end, we use SID to detect known vulnerabilities in the Linux kernel to evaluate how many of them are missed by SID. Specifically, like selecting recent patches in §3.1.2, we chose patches from 100 recent vulnerabilities, which violate at least one of our security rules.We used SID to analyze the corresponding patches. It turns out that SID found 47 vulnerabilities out of the 100. Therefore, SID missed 53% of vulnerabilities. After manually checking the false-negative cases, we found the following main causes. The corresponding solutions to reducing false negatives are discussed in §3.6.

Under-constrained symbolic execution. SID uses under-constraint symbolic execution in both patched code and unpatched code to determine security impacts

conservatively. In some cases, even when the patched code can never violate a security rule or the unpatched code must violate the security rule, the conservative execution may not be able to prove it. The conservative approach is mainly to reduce false positives, which, however, introduces a significant number of false negatives. This problem causes 17 cases.

Incomplete coverage for security and vulnerable operations. Intuitively, the incompleteness of the coverage will result in false negatives in confirming security impacts. In the current implementation, we collected the most common operations based on our statistical study (see §3.4.3). For example, for out-of-bound read or write bugs, we only collected the vulnerable operations such as array operations, common read and write functions; however, vulnerable operations using custom functions would be missed. This problem causes 31 cases. Also, similar to the causes of false positives, there are 5 cases caused by inaccurate static analysis.

3.5.3 The Trade-off between False Positive and False Negative

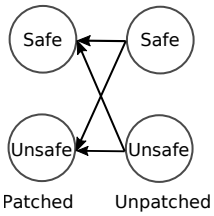

Figure 3.4: The safety-state transition diagram from unpatched version to patched version.

Handling partial-fix patches. To distinguish the root difference between patches for general bugs and security-related bugs, we introduce the concept of safety-state transition. First, we say that the program is in an unsafe state if it violates at least one security rule; we say the program is in a safe state if the violations are eliminated. Figure 3.4 shows all the transition states. Most commonly, a patch fixes a security bug if the unpatched version is in the unsafe state, and the patched version is in the safe state. In this case, we say that the patch blocks all the security impacts of the bug. However, in some corner cases, a patch only relieves a security bug, for which both patched and unpatched versions are in the unsafe state, and the patched version has fewer security-rule violations than the unpatched version does. Thus, we call them *partial-fix patches*. In this project, since

SID is designed to cover the patches that correctly and completely fix a security bug, it may miss security bugs with partial-fix patches. Table 3.7 summarizes how partial-fix can happen for the covered types of bugs. In the course of our evaluation, we only found two partial-fix cases—incompletely initializing memory, which shows that partial-fix patches are not common. However, in the future, it is possible to extend SID to detect such partial-fix cases by analyzing the security operations in a finger-grained manner (e.g., which bytes have been initialized) and relaxing the symbolic rules (i.e., does not require the block of all violations).

Bug types	**Partial fix on security operations**
Out-of-bound access	Incomplete bound check
Use-after-free/double-free	N/A
Uninitialized use	Incomplete initialize
Permission bypass	Incomplete perm check

Table 3.7: Possible partial-fix patches.

Relaxing symbolic rules. A unique strength of SID is using the conservativeness of under-constrained symbolic execution to precisely determine security impacts. Specifically, if the under-constrained symbolic execution is *unsolvable*, it is truly unsolvable; however, if the under-constrained symbolic execution is *solvable*, it can be a false positive due to missing constraints. Therefore, we design strict rules—the patch should block all violations (against a security rule) in the unpatched version by proving the opposite constraints (see §3.3.2) unsolvable.

Readers may wonder whether we can relax the symbolic rules—determining security impacts as long as some violations have been blocked—to reduce false negatives of SID. A critical issue with rule relaxing is that, with the relaxed rules, we cannot construct the opposite constraints to prove the unsolvability. This will prevent SID from benefiting from the conservativeness of under-constrained symbolic execution because the detected blocks of violations will likely be false positives due to missing constraints, rendering the security-impact determination highly imprecise.

3.5.4 Security Evaluation for Identified Security Bugs

Every patch identified by SID, besides the false positives (3%), fixed at least one security impacts; therefore, we believe that the corresponding bugs are security bugs. To

further validate that the identified security bugs are real vulnerabilities, we conduct two evaluations: (1) requesting CVEs and (2) analyzing reachability.

Vulnerability confirmation for CVE. Surprisingly, out of 227 security bugs found by SID, only 31 of them have been already assigned with CVE numbers. The remaining 196 security bugs were not reported and were improperly treated as non-security bugs. To confirm vulnerabilities, we request CVEs for the remaining security bugs in two phases. In the first phase, we requested CVEs for 40 security bugs individually. Due to that requesting CVE is a time-consuming process; in the second phase, we requested CVEs in a single batch.

The 40 security bugs submitted in the first phase come from two sets; the first set contains 21 bugs that are patched in the Linux kernel but still unpatched in the Android kernel. We believe that this set represents *less-likely* security bugs because the Android team might have confirmed them as non-security bugs thus did not patch them. The second set includes the other 19 detected security bugs that are randomly selected but cover different types of security impacts. For these 40 cases, which were submitted individually, we have received responses for 37 of them. In particular, we have obtained 24 CVEs for 23 security bugs (one bug was assigned with two CVE IDs), and 9 of these bugs are from the unpatched set in Android. 14 security bugs will not be assigned with CVEs due to non-technical or controversial reasons: (1) the security bug is in pre-release versions (5 cases); (2) information-leak (memory disclosure to userspace) bugs in obsolete kernel code [1] (5 cases); (3) patch commits do not mention security impacts (4 cases). For the first reason, we believe that most code will be released, and the corresponding security bugs will qualify CVEs. For the second reason, we would disagree—memory disclosures to userspace are security-critical because they break ASLR [102, 103] and leak sensitive data; also, the involved code still exists in the latest and released versions. The third reason shows that manually confirming the security impacts of bugs without commits mentioning security issues is hard.

Because maintainers would not assign CVEs for bugs in the pre-release (i.e., release candidate (RC)) versions, in the second phase, we filtered out 42 such bugs and reported 114 security bugs in a single batch. We still have not received the responses yet because

[1] Response: "CVE IDs are not required for information leak to userspace in various obsolete kernel code from approximately 2013."

the review of the reports for these bugs requires significant manual work for CVE maintainers. In comparison, individual reports receive responses much more timely. In summary, we totally reported 154 security bugs to CVE maintainers. We have received 37 responses with 24 new CVEs assigned. This means that, including the previously confirmed CVEs, 54 out of 227 identified bugs have been assigned with CVEs. Note that none of the rejected cases is due to misidentifying security impacts. Table A.4 show the details of the security bugs and CVE. These results indicate that SID is effective in determining security bugs from massive general bugs.

Reachability analysis for security bugs. Since a bug becomes a vulnerability when it can be triggered and cause security impacts, we also evaluate the reachability (from attacker-controllable entry points) of the identified bugs. The identified security bugs were detected through either fuzzers or other techniques such as static analysis. Clearly, if a bug was found by fuzzers such as Syzkaller [47], we can directly confirm its reachability. In particular, by checking the git commits, which would mention the corresponding fuzzers if a bug was found through fuzzing, we found that 28 (12.3%) of identified security bugs were found by fuzzers, thus are reachable from attackers.

The remaining 199 security bugs were mainly found through static analysis; confirming their reachability from attacker-controllable entry points has been a challenging and open problem [104]. Therefore, in this evaluation, we focus on finding the reachable call-chain between attacker-controllable functions and the functions containing the vulnerable operations. To this end, we first identified entry points—functions in the kernel that can be arbitrarily called by attackers. Based on how the kernel interacts with external entities, we empirically identify the following entry points.

- System calls. They are the most commonly used interface between user-space and kernel-space, which are also widely targeted by kernel fuzzers.
- Driver-specific I/O-control handlers. These handler functions are registered and can be called through the ioctl system call. By setting specific parameters, attackers can control the handlers. The previous work, DIFUZE [48], also fuzzes these handlers to find bugs in the kernel drivers.
- Interrupt (IRQ) handlers in drivers. Malicious hardware can invoke such handler functions by triggering the interrupt and prepare their parameter; therefore, they are also controllable to attackers. PeriScope [49] also fuzzes kernel drivers and regards

these IRQ handlers as entry points.

We first identify the 338 system calls in the Linux kernel based on the system-call list [105]. Then, following the method of DIFUZE [48], we identify a set of structures that can be used to register ioctl handler by drivers, and based on these structures, we find 603 ioctl handlers [106]. To identify IRQ handlers, PeriScope [49] shows that drivers can register their own IRQ using multiple types of APIs, and tasklest is one of the most commonly used software interrupts (softirq). Therefore, based on the declarations of tasklest and IRQ-related keywords in drivers, we finally find 126 IRQ handlers.

The idea of the evaluation is to traverse the global call-graph of the kernel to collect the shortest call-chain path between the entry points and functions containing the vulnerable operations. We employ Dijkstra's Shortest Path (DSP) algorithm [107] to find the paths. Given a bug, we say it is reachable from attacker-controllable entry points if we find such paths. To minimize false positives and false negatives, we employ the state-of-the-art techniques—using struct types to match functions [100, 99, 108]—to precisely identify indirect-call targets. Table 3.8 shows the number of bugs that are reachable from different types of entry points. In particular, we found that 133 security bugs are reachable from systems, and 154 are reachable from the three classes of entry points.

Entry points	**Num of reachable bugs**
Dynamically confirmed bugs (fuzzers)	28
System calls	133
I/O control handlers	148
Interrupt handlers	131
Total	154 (67.8%)

Table 3.8: Number of bugs that can be reached from different kinds of entry points.

3.5.5 Generality of SID's Patch Model

SID's patch model includes three key components of patches: security operation, vulnerable operation, and critical variables. To evaluate the generality of the model, we analyzed the most recent 100 vulnerabilities in the Linux kernel that were disclosed in 2019. Table 3.9 shows the statistical results of this evaluation.

We can find all of the three components in 77 vulnerabilities; therefore, SID can support the security-impact determination for them. Furthermore, 11 vulnerabilities only

Num of key components	Percent
Three components	77%
Two or one component	11%
Other cases	12%

Table 3.9: The generality of SID's patch model. It shows the numbers of components the vulnerabilities have.

have one or two of these key components. For example, pointer usage in an incorrect order can introduce use-after-free vulnerabilities, and the corresponding patches just change the pointer reference order. In this case, the security operation is not modeled and thus will be missed. In addition, 12 cases involve code removal as the fix or multiple patches, which cannot be clearly represented by SID's current model. For example, there are five patches that only delete some redundant code, such as deallocation functions. Some vulnerabilities were fixed by more than one patches or complex patches.

SID's model is, in fact, conceptual and general—while a vulnerability typically has vulnerable operations, the patch performs security operations to prevent them, and both kinds of operations often target variables. In the future, we can certainly extend the model to support more cases. For example, even for memory leaks where there is no explicit vulnerable operation at all, we can artificially model "object pointer never being released" as the vulnerable operation, which can be realized by analyzing the operations against the pointer (critical variable).

3.5.6 New and Important Findings

Patching-time window for security and general bugs. In order to show the importance of SID, we would like to know how Linux maintainers treat security bugs and general bugs differently. To this end, we measure and compare the patching time window for them—the time from the submission/report of a patch to the application of the patch. We tested 8,000 patches for general bugs, 1,339 patches for vulnerabilities, and all the security bugs that are found by SID but do not have CVE ID. We use the cumulative distribution function (CDF) to show the statistical patching-time window in Figure 3.5. We find that the patching-time window for CVE-assigned vulnerabilities (5.8 days) is shorter than security bugs (8.6 days) found by SID. This means that the maintainers have not treated these security bugs as important as vulnerabilities.

We also find that the patching time window of security bugs identified by SID is shorter than other general bugs. We believe one reason is that the patches for these security bugs have fewer code changes, and the bugs have clearer patterns, which is also reported by Li et al. [109].

More statistical results are shown in Table 3.10, from which we can find that security patches still take a long time to be applied. Nearly 10% of security bugs are patched more than one month after they have been reported. This significant time window gives attackers much time to craft critical exploits, not to mention that the reported patches are visible to attackers. Thus, an automated tool that can determine the security impacts of bugs is demanded.

Type	**Average (Days)**	**Median (Days)**	**Maximum (Days)**
General patches	15.8	3	1240
Patches of security bugs	8.6	2	111
Patches of vulnerabilities	5.8	1	974

Table 3.10: Statistics on patch-time window.

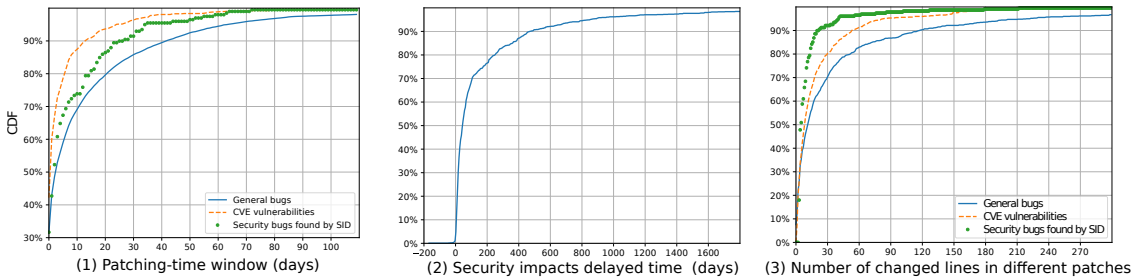

Figure 3.5: Statistical findings. CDF: cumulative distribution function; (1) CDF for time windows from bug report date to bug fix date; (2) CDF for time window from the patch date to the CVE release date; (3) CDF for the number of changed lines for different kind of bugs. In (2), about 1% of vulnerabilities are assigned with a CVE before the bugs are actually patched.

Delayed disclosure of security impacts of existing vulnerabilities. We found that the disclosure of the security impacts of existing vulnerabilities is commonly delayed, which is also known as hidden impact vulnerability [81]. To measure the delaying, we define the delayed time as the time window from the patch date to the CVE-release date. We collected 1,339 vulnerabilities in the Linux kernel from the CVE database [110, 111]

and analyzed the delayed time. The CDF for the delayed time is shown in Figure 3.5. The results show that only 23.9% of them are identified as vulnerabilities less than two weeks (14 days) after they have been patched. The other 75% are reported as vulnerabilities after two weeks. The average and median delayed time period for these vulnerabilities is 191 and 45 days. The longest case is more than 12 years, which was identified as a security bug by SID and assigned with a CVE (CVE-2007-6762) after we reported it.

Security bugs threatening derivative software. Many programs are derived from other open-sourced programs. For example, the Android kernel is a modified version of the Linux kernel, and the Android kernel is further customized into thousands of versions [112] running on tens of thousand device models [113]. This problem is known as Android fragmentation [113, 112]. Manufactures are unable to fix all bugs timely, due to a large number of derivative programs. Instead, they prioritize security-critical ones and postpone or even ignore non-security ones. To understand the severity of the problem, we test more than 5K bug patches in the Android kernel of version 4.14-p-release and perform two evaluations.

The first evaluation is to check how many security bugs (identified by SID) remain unpatched in Android. Specifically, we manually checked the security bugs found by SID in the latest (as of the experiment) Android kernel release version, Android-4.14-p-release, which was ported from the Linux kernel 4.14 on *November 12, 2017* [114]. As such, patches applied before the date in Linux will also be available in Android. Therefore, we analyzed the security bugs found by SID that were introduced before November 12, 2017, but patched in Linux after the date. We found that 39 such security bugs were reported after November 12, 2017, in the Linux kernel and were not assigned with a CVE; 11 of them do not affect the Android code anymore; thus, they are excluded. For the remaining 28 security bugs, we found that only seven are patched in the Android kernel, and 21 (75%) remain unpatched. Details can be found in appendix (Table A.4). These security bugs may pose serious security risks to Android.

The second evaluation is to measure the bug-fixing time windows of CVE-assigned bugs and non-security bugs in Android. Specifically, non-CVE bugs are fixed with an average of 44.6 days (30 days following Linux kernel patches), while CVE-assigned bugs are fixed with an average of 27.8 days (only 14 days following the Linux kernel patches). For security bugs, because most of them are not identified as vulnerabilities, thus they

will be treated as general bugs and would not be fixed in time. The average time window is 44.6 days. For instance, Figure 3.6 is a missing bound check security bug in the Linux kernel. After our report, this security bug has a CVE ID, CVE-2019-15926, with CVSS score 9.4. This bug was introduced from the Linux kernel 3.0 but still has not been patched in the Android 4.14-p-release until the submission of this paper.

```
/*drivers/net/wireless/ath/ath6kl/wmi.c
 * CVE-2019-15926, CVSS 9.4*/

static int ath6kl_wmi_pstream_timeout_event_rx(
                struct wmi *wmi, u8 *datap,
                int len) {
    ...
    ev = (struct wmi_pstream_timeout_event *) datap;
+   if (ev->traffic_class >= WMM_NUM_AC) {
+       ath6kl_err("invalid traffic class: %d\n",
+               ev->traffic_class);
+       return -EINVAL;
+   }

    ...
    wmi->stream_exist_for_ac[ev->traffic_class] = 0;
    ...
}
```

Figure 3.6: An out-of-bound access vulnerability in Android 4.14

Multi-impacts per bug. The semantics of vulnerable operations often decides security impacts. Intuitively, when a critical variable is used in multiple vulnerable operations, it may have multiple security impacts. In particular, for the 227 security bugs, SID actually found 243 security impacts, as shown in Table 3.11. For example, some out-of-bound access cases are also caused by uninitialized use when the uninitialized variable is used as a size variable in memory access.

Security impacts	#	**Security impacts**	#
Uninitialized use	85	Use-after-free/Double-free	67
Out-of-bound access	65	Permission bypass	13
NULL-pointer dereference	13		

Table 3.11: Security impacts for the identified security bugs

Characterizing bugs and vulnerabilities. In addition to the security impacts, we also characterize other differences between general bugs and security bugs. First, we analyzed

the differences in the number of changed lines in patches for 1,350 randomly selected general bugs, the security bugs found by SID, and 1,339 CVE-assigned vulnerabilities. The statistical results are shown in Figure 3.5. We can find that security bugs and vulnerabilities are highly similar in terms of the number of changed lines in their patches—both have a smaller number then general bugs patches have. The finding is consistent with the results found by Li et al. [109]. Also, the patches for vulnerabilities and security bugs tend to change fewer files than patches for general bugs. Patches for general bugs changed 3.0 files on average, which in contrast to 2.3 files for patches of vulnerabilities and security bugs found by SID.

Another finding is that among the git commits for the 1,339 vulnerabilities, 814 (60.8%) of them did not mention any security impacts such as use-after-free, double-free, etc. This result implies that one cannot reliably determine the security impacts solely based on the textual information in patch commits.

3.6 Discussion

The generality of SID. SID can be extended to determine the security impacts of patches for other well maintained open-source programs such as Firefox, Chrome, and FreeBSD. Our patch model is general, which can be used to describe the security operations and vulnerable operations in a program-agnostic manner. To extend SID to other programs, only the pre-processing part requires new manual effort. For example, for out-of-bound access vulnerabilities, for different target programs, we need to collect functions for memory access and bound checks.

The extensibility of SID. In our work, SID is only used to support the common classes of vulnerabilities. However, other classes of vulnerabilities can also be supported by specifying the security rules for them—how these vulnerabilities violate the security rules. In addition, the rules for constructing constraints from security operations and security rules should also be specified. Table A.5 shows that we can naturally extend SID to support more classes of bugs by modeling the three components of their patches. For example, numerical-error vulnerabilities such as divide-by-zero can be supported. The security rule is that the divisor cannot be zero. Correspondingly, the vulnerable operation is division, and the security operation can be a zero-check for the divisor.

Similarly, NULL-pointer dereference also fits SID's model. The security rule is that a dereferenced pointer cannot be NULL. Therefore, the vulnerable operation is pointer dereferencing, and the security operations can be a NULL check. After including these types, Sid can cover at least 51% of vulnerabilities (13% more).In the future, we would like to support more classes of vulnerabilities.

In addition, the current implementation of SID considers only simple patching patterns for vulnerabilities because we find that the average distance between a security operation and a vulnerable operation is 6.6 lines of code, and the longest distance is 65 lines of code. This result is consistent with the finding of SPIDER [115]—94.4% of safe patches affect less than 20 lines of code. Therefore, the under-constrained symbolic execution can handle most of them efficiently. In the future, SID can also be extended to support complicated patterns. For example, some patches use multiple security operations together to fix a vulnerability. These cases can be described using complex security rules and represented with multiple constraints.

Reducing false negatives. First, the conservativeness of the under-constrained symbolic execution indeed introduces a significant number of false negatives, because in the current implementation, the constraints for the slice paths are collected only from the security operation to the vulnerable operation. Therefore, the constraints are under-constrained. As an improvement, in the future, we can extend the constraint collection beyond the security operation—backwardly collecting as many constraints as possible from the security operation and adding them to the final constraint set. This method could reduce at most 17% of false negatives in our evaluation.

Second, the incompleteness of the security and vulnerable operations sets also causes false negatives. To reduce them, we can collect more custom functions for security and vulnerable operations. Such functions can be modeled based on dynamic analysis [116] and wrapper-function analysis [100, 117]. Covering more security and vulnerable operations can reduce at most 31% of false negatives. For example, if we model the lock/unlock operations for use-after-free, we can additionally cover 25% of use-after-free vulnerabilities in the evaluation. But to do so, more manually analysis work on patches and engineering efforts are needed. Therefore, we put these works in future works.

3.7 Related work

Mining security-critical vulnerabilities from bugs. Wijayasekara et al. [81] show the hidden impact vulnerabilities that were first identified as non-security bugs and publicized and later were identified as vulnerabilities due to exploits. In addition, previous work [79, 80, 82, 84, 118, 83] has used supervised and unsupervised learning techniques to classify the vulnerabilities and general bugs based on the textual information of the patches. Tyo [82] showed that the Naive Bayes and Support Vector Machine classifiers always have the best performance. However, such work cannot handle the patches without descriptions or if they have incomplete/inaccurate descriptions. Moreover, these works focused only on differentiating vulnerabilities from general bugs, which cannot determine the specific security impacts of the bugs or pinpoint the vulnerable operations resulting in the security impacts. Recent work, SPIDER [115], identifies fixes as security fixes as long as they do not disrupt the intended functionalities. This assumption does not hold for some patches, such as the ones only improving code readability or replacing equivalent APIs. In contrast to these papers, SID can precisely determine the security impacts of a patched bug and provide details on the vulnerable operations even when the commit description is not available. Moreover, SID is not based on the assumptions used by SPIDER.

Testing the exploitability of bugs. Several prior studies have attempted to test the exploitability of a particular class of bugs. Specifically, Lu et al. [119] showed how to exploit uninitialized-use bugs using symbolic execution and fuzzing in the Linux kernel. Xu et al. [52] presented a memory collision strategy to exploit the use-after-free vulnerabilities in the Linux kernel. You et al. [93] presented SemFuzz, which guides the automatic generation of proof-of-concept exploits for vulnerabilities. Thanassis et al. [120] proposed AEG, a symbolic execution–based automatic exploit generation tool that can automatically exploit memory-corruption bugs such as buffer overflow. Wu et al. [94] presented FUZE, which can automatically exploit use-after-free vulnerabilities in the Linux kernel. Unlike these studies, SID does not focus on the exploitability of a specific class of vulnerability. Instead, SID aims to automatically determine the security impacts of a bug once it is triggered. Moreover, SID is not limited to a specific vulnerability class.

Bug-severity assessment. Mell et al. [121] presented the common vulnerability

scoring system (CVSS), which is the most widely used vulnerability scoring system. CVSS requires manual scoring of the severity of vulnerabilities based on their confidentiality, integrity, and availability. However, Munaiah et al. [122] showed that CVSS is often biased in determining the severity. For example, it does not treat code execution and privilege escalation as important factors when analyzing the severity of vulnerabilities. Most of the severity-analysis techniques [123, 124, 125, 126] are based on bug reports, which also cannot handle the bug patches without a description or if they have incomplete descriptions.

Symbolic execution. Symbolic execution has been used for decades. Cadar et al. [127] proposed a symbolic execution method to generate inputs to trigger bugs in real code automatically. Later, Cadar et al. developed KLEE [128], which is a widely-used symbolic execution engine. Both of these tools need the complete constraints in the program; thus, they can only symbolically execute from the entry of a program. Such symbolic execution does not scale well to large programs. Under-constrained symbolic execution [129, 95], implemented in UC-KLEE, lifts this limitation by treating symbolic values coming from unexecuted parts of the code especially. This approach makes symbolic execution much more flexible and expands the possible applications. Because under-constrained symbolic execution is unaware of properties of data established by the unexecuted code, it can still produce false-positive error reports that would not occur when executing a complete program. Similar to UC-KLEE, SID also uses under-constrained symbolic execution to execute from an arbitrary point in a function symbolically. However, SID minimizes false positives by combining the constraints from security rules and differential analysis. Also, different from UC-KLEE, which detects bugs introduced by new patches, SID determines the security impacts of patches.

Chapter 4

DiffCVSS: Automatically Identify CVSS-related Functionality-Semantics to Facilitate Vulnerability Prioritization

Linux has become the most widely used and complex open-source project. The Linux kernel not only evolves quickly, but is also commonly cloned and customized, which results in a large number of versions and derivatives. Specifically, it has more than three thousands of different versions, including stable versions, release candidate versions, and long time support versions. Many of them are commonly used by the systems such as Android, Ubuntu, Red Hat, and IoT systems are also derived from the Linux kernel. For example, there are at least 29 [130] major Android systems running on over 24,000 models [131] and billions of mobile devices.

The Linux kernel alone is reported to have thousands of bugs each year, and hundreds of them are security related bugs (vulnerabilities). When a vulnerability is severe, it is supposed to be patched promptly to avoid being exploited [132]. This is crucial given its extremely high importance and popularity. To assist with the severity assessment, maintainers widely use Common Vulnerability Scoring System (CVSS) [133], an open framework for characterizing the severity of vulnerabilities. CVSS is a metric-based

system; combining all CVSS metrics with different weights allows people to calculate [10] a score ranging from 0 to 10 (the most severe). In practice, CVSS has been widely adopted as a standard measurement system by industries, organizations, and governments; the National Vulnerability Database (NVD) provides CVSS scores for almost all known Linux vulnerabilities.

The "one for all" CVSS usage. A fundamental problem arises when CVSS meets Linux—it is used in an "one for all" manner. When a bug reporter requests a *Common Vulnerabilities and Exposures* (CVE) [134] for a vulnerability, the CVE maintainers assign a (single) CVSS score for it, typically based on the mainstream Linux. All affected versions and some derivatives will then simply honor the assigned CVSS score for prioritizing their patches. This is understandable because assigning the CVSS score is quite laborious and requires expertise. Maintainers of small derivatives may not afford the reevaluation for all of their system.

A "one for all" CVSS usage results in two critical problems in vulnerability prioritization. First, patches for a severe vulnerability may be delayed or even ignored when its severity is underestimated. While a CVSS score is assigned for the project where the vulnerability was originally found, the vulnerability may manifest a much higher severity in a different version or derivative. Second, overestimating the severity in a different version or a derivative may waste maintenance resources when they are improperly allocated to non-critical vulnerabilities, which delays the patching for more critical vulnerabilities.

The severity of vulnerabilities varies significantly across different operating systems (OSes) [135]. In recent years, some security-sensitive vendors (e.g., Red Hat [136], Ubuntu [137] and BlackBerry [138]) have begun publishing their own severity levels for vulnerabilities that affected their products. For example, Red Hat re-evaluated CVSS scores of 2,199 Linux-related CVEs, among which 981 CVEs have different CVSS scores from the original ones, with 247 having higher scores in Red Hat. Unfortunately, such *OS-aware* severity analysis is commonly done manually by analyzers [139] and only by major vendors. Such an approach certainly would not scale and be affordable for small vendors. As a result, reusing the CVSS scores assigned by NVD is still the dominating strategy in practice. Hence, it is essential to automatically analyze the severity of vulnerabilities in an OS-aware manner, to support thousands of affected derivatives and

versions.

OS-aware vulnerability prioritization: challenges. Given a vulnerability, automatically calculating its CVSS score for different OSes is challenging. First, determining the exploitability of a vulnerability is still an open problem [140], which requires understanding code semantics, reachability, environments, etc. Second, CVSS involves many metrics from multiple dimensions. Automatically assessing them to determine the scores is hard. To our knowledge, none of the existing works can provide *OS-aware* and automated severity analysis for thousands of derivatives and versions of a program like the Linux kernel.

Our approach. In this paper, we propose OS-aware vulnerability prioritization (namely DiffCVSS) for Linux-based systems, which employs differential severity analysis for Linux derivatives and versions. Specifically, given a Linux CVE (i.e., a vulnerability assigned with a CVSS score for the mainstream Linux), DiffCVSS employs both static program analysis and natural language processing (NLP) to precisely identify and map Linux functions to CVSS metrics, and match code paths related to the CVE in both the mainstream version and the target version. It then performs OS-aware analysis for the metrics-related functions in the code paths. By differentially comparing the metric-related functions, DiffCVSS automatically determines if the vulnerability is less or more severe in the target version. DiffCVSS pinpoints such cases for maintainers to further reevaluate the severity for the specific target version. A unique strength of this approach is that it transforms the challenging CVSS calculation into automatable differential analysis. More specifically, to realize DiffCVSS, we propose multiple new techniques.

First, we identify CVSS-related functions and map the CVSS metrics to them. The technique trains a set of classifiers using the Bi-directional Long Short-Term Memory Networks (BiLSTM) [141] +attention model. We choose this model because it can capture the semantic context of a full sentence, also pay more attention to those informative words that have significant impact to classification results. It further leverages transfer learning to transform semantic knowledge to a specific domain. Second, we identify and map call-chains (vulnerability paths) for a CVE. This technique employs both static program analysis and NLP techniques to precisely locate and match the call-chains in Linux and its derivatives. Third, we perform metric-level differential analysis against functions in the call-chains and determine if the vulnerability deserves a severity reevaluation in the

target OS version.

We have implemented DiffCVSS and applied it to the mainstream Linux and downstream Android systems. We choose them because they represent the most popular Linux-based ecosystem. We found that DiffCVSS is able to precisely map CVSS-related functions and identify the call-chains leading to the vulnerability. More importantly, with DiffCVSS, we found 110 vulnerabilities that have different severity levels between Android and Linux, and 30 vulnerabilities that have different severity levels across different versions of the Linux kernel itself. In 18 cases, the severity is much higher in the derivative Android system. Failure to re-assess them would delay the patching of severe vulnerabilities, which incurs significant threats. These results show that DiffCVSS offers a precise and effective way to identify vulnerabilities that manifest different severity levels in a specific OS and thus deserve a severity reevaluation. In addition, we conduct a user study on DiffCVSS, and the results demonstrate the effectiveness and usability of DiffCVSS for its users (e.g., maintainers).

4.1 Background

Table 4.1: Examples of re-evaluated CVEs by different vendors. N: Number of vendors; VS: Vendor Severity; NS: NVD Severity; L: Low; M: Medium; H: High; C: Critical;

System Type	N	Vendor	CVE	VS	NS
Mobile devices	5	BlackBerry, Huawei, LG,etc.	CVE-2020-11652	M	H
IoT/ICS devices	7	NetApp, Siemens, SAP, etc.	CVE-2018-2477	H	M
Network devices	8	Cisco, PulseSecure, SonicWall etc.	CVE-2020-1993	M	L
Personal computer	6	Ubuntu, Red Hat, SUSE etc.	CVE-2017-5897	L	C

4.1.1 Cross-OS Vulnerabilities

A vulnerability becomes a cross-OS vulnerability when it exists in many OSes (e.g., Linux, Android, and Red Hat) and causes a different severity in them. Such vulnerabilities should be evaluated separately per OS.

Prevalence of cross-OS vulnerabilities. The Linux kernel has been shipped to a wide variety of computing systems, such as IoT devices, mobile devices (mainly Android), personal computers, and industrial control systems (ICS). One of the most well-known Linux derivatives is the Android common kernels [142], also known as ACKs, which

are downstreams of the Linux kernel. Furthermore, plenty of mobile OSes are based on ACKs or Linux kernel, such as BlackBerry Secure [138], ColorOS [143], EMUI [144], MIUI [145], and Chrome OS [146]. Therefore, most vulnerabilities in Linux and Android are cross-OS vulnerabilities. Our study on 2,911 CVEs in the Linux kernel and 6,080 CVEs in Android found that 26 vendors and 10 third parties have reevaluated the severity of these vulnerabilities on their own or other platforms. Table 4.1 shows several example vulnerabilities that were reevaluated by different vendors. Although some major vendors have their own criteria for reevaluating the severity [136, 137, 147]. The criteria are rough and hard to automate for analyzing different vulnerabilities. These results indicate that cross-OS vulnerabilities are pervasive and have raised awareness in major vendors (but not in small vendors yet).

4.1.2 Impacts of Cross-OS Vulnerabilities

Table 4.2: Delayed patch days of the Linux vulnerability on Android-MSM project [3]. DD = delay days.

Linux Severity (CVSS 3.0)	Medium of DD	Average of DD
LOW	349	349
MEDIUM	99	138.5
HIGH	34.5	57
CRITICAL	8	8
Average	122.6	138.5

The severity of a vulnerability would significantly influence the patch prioritization of the vulnerability. However, in practice, the "one for all" strategy is widely adopted, regardless of the underlying OSes, which would inevitably result in overestimation or underestimation of the severity. We next present how overestimation and underestimation result in security concerns.

Underestimation causes delayed or even missed patches, leaving the program vulnerable. Given the limited maintenance resources, software vendors have to de-prioritize the patching of vulnerabilities with lower severity level. Android Security [132] mentions that *"The first task in handling a security vulnerability is to identify the severity of the bug and which component of Android is affected. The severity determines how the issue is prioritized."* When comparing the patch time of vulnerabilities in the Linux kernel

and the Android-MSM project [3] (see Table 4.2), we found that the delays (in days) are inversely proportional to the severity reported by the NVD. Therefore, underestimation of vulnerability can lead to a delay of months and even years. In practice, we have actually observed many cases where underestimation leads to delayed and missed patches, such as CVE-2016-5696 [148]. It is worth noting that when a vulnerability is assigned with a CVE, it has been publicized, which means adversaries know them. In this case, delaying or ignoring the patches is particularly critical.

Overestimation wastes limited maintenance resources (which in turns also delays the patching for more critical ones) and is quite common. According to the data released by NVD [149], from 2017 to 2020, the number of vulnerabilities disclosed each year has almost doubled from that before 2016. On average, each enterprise will find 870 CVEs from 960 IT assets every day [150], and they usually follow the severity scores published on NVD. Specifically, 33.4% of vulnerabilities re-evaluated by Redhat have a lower CVSS score than that from NVD. Hence, handling a large amount of vulnerabilities a day poses a big challenge for organizations, especially on those overestimated, which will lead a serious resource drain. Such inappropriate and non-optimal resource allocation could in turn result critical vulnerabilities in being delayed.

Overall, the "one for all" CVSS usage can cause many issues, and more and more vendors have started to re-evaluate the vulnerabilities. CVSS scores are widely used to prioritize the fixes of vulnerabilities. The timeliness of the patching of a vulnerability is often proportional to its CVSS score [151]. CVSS scoring has been complained of being too generic by lots of organizations [152], without considering different execution contexts [153], which are common due to customization. As a result, more and more vendors started performing the re-evaluation of vulnerabilities for proper prioritization and risk management. We found that their re-evaluation results are alarming—severity differences are quite prevalent in cross-OS vulnerabilities; nearly 44.6% of vulnerability scores re-evaluated by Red Hat are different from that of NVD. It is also worth noting that existing re-evaluation is largely manual [139], which cannot scale and would still slow down the patching by months or even years [151]. Furthermore, in our user study (see §4.7), almost all participants agreed that the "one for all" CVSS usage is problematic, and re-evaluating severity is necessary but laborious, time-consuming, and expertise-required.

4.1.3 CVSS Metrics

The CVSS is an open and widely-adopted vulnerability severity scoring standard. It assigns severity scores to vulnerabilities, which allows responders to prioritize resources for responses. A vulnerability is typically assigned a CVSS score rating from zero to ten that maps to severity levels from low to high (i.e., 0.1-3.9 as low, 4.0-6.9 as medium, 7.0-8.9 as high, and 9.0-10 as critical). To generate a CVSS score, an assessor will follow the CVSS specification document [133] to assign values to a set of metrics. A CVSS score is then calculated according to a CVSS vector aggregating CVSS metric values. The formulas can be found in [133]. The CVSS score influences the enthusiasm of applying patches from relevant product suppliers.

Linux has become the most widely used and complex open-source project. The Linux kernel not only evolves quickly, but is also commonly cloned and customized, which results in a large number of versions and derivatives. Specifically, it has more than three thousands of different versions, including stable versions, release candidate versions, and long time support versions. Many of them are commonly used by the systems such as Android, Ubuntu, Red Hat, and IoT systems are also derived from the Linux kernel. For example, there are at least 29 [130] major Android systems running on over 24,000 models [131] and billions of mobile devices.

The Linux kernel alone is reported to have thousands of bugs each year, and hundreds of them are security related bugs (vulnerabilities). When a vulnerability is severe, it is supposed to be patched promptly to avoid being exploited [132]. This is crucial given its extremely high importance and popularity. To assist with the severity assessment, maintainers widely use Common Vulnerability Scoring System (CVSS) [133], an open framework for characterizing the severity of vulnerabilities. CVSS is a metric-based system; combining all CVSS metrics with different weights allows people to calculate [10] a score ranging from 0 to 10 (the most severe). In practice, CVSS has been widely adopted as a standard measurement system by industries, organizations, and governments; the National Vulnerability Database (NVD) provides CVSS scores for almost all known Linux vulnerabilities.

The "one for all" CVSS usage. A fundamental problem arises when CVSS meets Linux—it is used in an "one for all" manner. When a bug reporter requests a *Common*

Vulnerabilities and Exposures (CVE) [134] for a vulnerability, the CVE maintainers assign a (single) CVSS score for it, typically based on the mainstream Linux. All affected versions and some derivatives will then simply honor the assigned CVSS score for prioritizing their patches. This is understandable because assigning the CVSS score is quite laborious and requires expertise. Maintainers of small derivatives may not afford the reevaluation for all of their system.

A "one for all" CVSS usage results in two critical problems in vulnerability prioritization. First, patches for a severe vulnerability may be delayed or even ignored when its severity is underestimated. While a CVSS score is assigned for the project where the vulnerability was originally found, the vulnerability may manifest a much higher severity in a different version or derivative. Second, overestimating the severity in a different version or a derivative may waste maintenance resources when they are improperly allocated to non-critical vulnerabilities, which delays the patching for more critical vulnerabilities.

The severity of vulnerabilities varies significantly across different operating systems (OSes) [135]. In recent years, some security-sensitive vendors (e.g., Red Hat [136], Ubuntu [137] and BlackBerry [138]) have begun publishing their own severity levels for vulnerabilities that affected their products. For example, Red Hat re-evaluated CVSS scores of 2,199 Linux-related CVEs, among which 981 CVEs have different CVSS scores from the original ones, with 247 having higher scores in Red Hat. Unfortunately, such *OS-aware* severity analysis is commonly done manually by analyzers [139] and only by major vendors. Such an approach certainly would not scale and be affordable for small vendors. As a result, reusing the CVSS scores assigned by NVD is still the dominating strategy in practice. Hence, it is essential to automatically analyze the severity of vulnerabilities in an OS-aware manner, to support thousands of affected derivatives and versions.

OS-aware vulnerability prioritization: challenges. Given a vulnerability, automatically calculating its CVSS score for different OSes is challenging. First, determining the exploitability of a vulnerability is still an open problem [140], which requires understanding code semantics, reachability, environments, etc. Second, CVSS involves many metrics from multiple dimensions. Automatically assessing them to determine the scores is hard. To our knowledge, none of the existing works can provide *OS-aware* and automated

severity analysis for thousands of derivatives and versions of a program like the Linux kernel.

Our approach. In this paper, we propose OS-aware vulnerability prioritization (namely DiffCVSS) for Linux-based systems, which employs differential severity analysis for Linux derivatives and versions. Specifically, given a Linux CVE (i.e., a vulnerability assigned with a CVSS score for the mainstream Linux), DiffCVSS employs both static program analysis and natural language processing (NLP) to precisely identify and map Linux functions to CVSS metrics, and match code paths related to the CVE in both the mainstream version and the target version. It then performs OS-aware analysis for the metrics-related functions in the code paths. By differentially comparing the metric-related functions, DiffCVSS automatically determines if the vulnerability is less or more severe in the target version. DiffCVSS pinpoints such cases for maintainers to further reevaluate the severity for the specific target version. A unique strength of this approach is that it transforms the challenging CVSS calculation into automatable differential analysis. More specifically, to realize DiffCVSS, we propose multiple new techniques.

First, we identify CVSS-related functions and map the CVSS metrics to them. The technique trains a set of classifiers using the Bi-directional Long Short-Term Memory Networks (BiLSTM) [141] +attention model. We choose this model because it can capture the semantic context of a full sentence, also pay more attention to those informative words that have significant impact to classification results. It further leverages transfer learning to transform semantic knowledge to a specific domain. Second, we identify and map call-chains (vulnerability paths) for a CVE. This technique employs both static program analysis and NLP techniques to precisely locate and match the call-chains in Linux and its derivatives. Third, we perform metric-level differential analysis against functions in the call-chains and determine if the vulnerability deserves a severity reevaluation in the target OS version.

We have implemented DiffCVSS and applied it to the mainstream Linux and downstream Android systems. We choose them because they represent the most popular Linux-based ecosystem. We found that DiffCVSS is able to precisely map CVSS-related functions and identify the call-chains leading to the vulnerability. More importantly, with DiffCVSS, we found 110 vulnerabilities that have different severity levels between

Android and Linux, and 30 vulnerabilities that have different severity levels across different versions of the Linux kernel itself. In 18 cases, the severity is much higher in the derivative Android system. Failure to re-assess them would delay the patching of severe vulnerabilities, which incurs significant threats. These results show that DiffCVSS offers a precise and effective way to identify vulnerabilities that manifest different severity levels in a specific OS and thus deserve a severity reevaluation. In addition, we conduct a user study on DiffCVSS, and the results demonstrate the effectiveness and usability of DiffCVSS for its users (e.g., maintainers).

In summary, this paper makes the following contributions:

• **Mapping functions to CVSS metrics.** We train a set of classifiers to map functions to the CVSS exploitability metrics based on their descriptions in Linux kernel and further leverage transfer learning to transform the semantic knowledge learned from Linux to the Android domain.

• **Identifying and matching vulnerability paths for CVEs.** Based on CVE information, DiffCVSS employs static program analysis and NLP to precisely identify the corresponding vulnerability paths (from an entry point to the vulnerable function) and match them between Linux and Android. We believe that identifying vulnerability paths is a useful technique that can enable further research such as patch generation and testing, and impact analysis.

• **OS-aware vulnerability prioritization.** With the mapping from functions to CVSS metrics and the identified vulnerability paths, DiffCVSS employs differential severity analysis, which can automatically determine the severity differences for the vulnerability in different OSes.

• **A severity reevaluation of Linux vulnerabilities.** With the new techniques, DiffCVSS achieves an impressive precision in the differential severity analysis. With DiffCVSS, we also found 110 vulnerabilities that have different severity across Android and Linux. More critically, 18 of them have a higher severity and should be reevaluated per OS to avoid delayed patching. Also, the usability study shows that DiffCVSS can guide maintainers to assess vulnerability correctly and effectively in an OS-aware manner.

There are two types of metrics, *exploitability metrics* and *impact metrics*. The exploitability metrics reflect the properties of the vulnerability that lead to a successful attack, and their values indicate the exploitation difficulty [133], which include the

following four parts: (1) *attack vector* (`AV`), reflecting the context in which vulnerability exploitation is possible. It consists of four values: `N` (network), `A` (adjacent network), `L` (local) and `P` (physical). (2) *attack complexity* (`AC`), indicating the additional conditions for a successful exploit; if a successful exploitation requires some measurable amount of efforts the `AC` should be `H` (high); otherwise, it should be `L` (low). (3) *privileges required* (`PR`), describing the privileges required for an exploit, ranging from `H` (high privilege requirement such as "root") to `N` (none privilege is required, thus easier exploitation). (4) *user interaction* (`UI`), showing the requirements for the user to participate in the exploitation, ranging from `R` (required, thus a harder attack) to `N` (none, thus an easier attack).

This project aims at analyzing exploitability metrics, instead of impact metrics (e.g., confidentiality, integrity, availability). This is because impact metrics are typically decided by the type of the vulnerabilities, and the associated impact score would not change across OSes. That said, based on the needs of vendors, the techniques proposed in this work can also be naturally extended to include impact metrics.

4.2 Overview of DiffCVSS

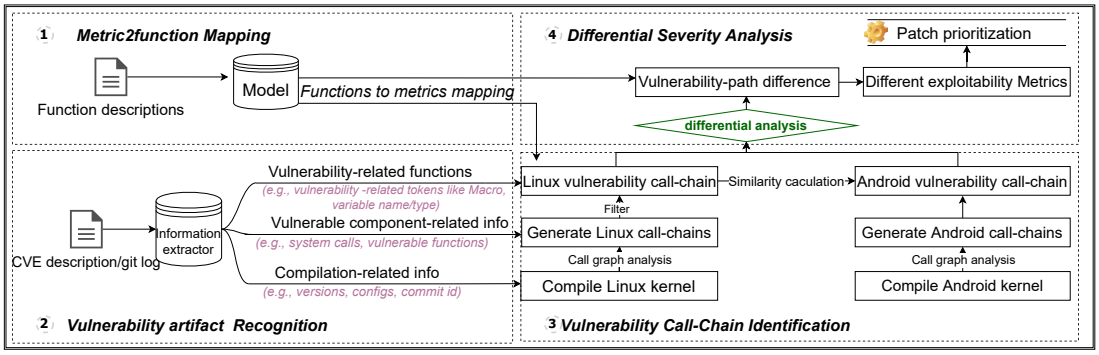

Figure 4.1: An overview of DiffCVSS.

DiffCVSS's goal is to enable OS-aware vulnerability prioritization. DiffCVSS employs differential analysis to automatically identify whether a vulnerability would manifest a different severity in a different OS. In this work, we focus on the most commonly used systems, the Linux kernel, and the derivative Android kernel. Their security can influence billions of devices. Figure 4.1 shows the overview of DiffCVSS, which consists of four parts: ① metric2function mapping, ② vulnerability artifact recognition, ③ vulnerability

call-chain identification, and ④ differential severity analysis. More specifically, using Linux/Android kernel function descriptions in the documentation, DiffCVSS constructs a map between functions and exploitability metrics (i.e., `AV`, `AC`, `PR`, `UI`) to support vulnerability severity quantification. For example, the Linux kernel function ns_capable with description *"determine if the current task has a superior capability in effect"* should be mapped with PR:H, indicating a high privilege requirement (①). Meanwhile, given a vulnerability in the Linux kernel, as documented by CVE, our approach extracts useful semantic information about the vulnerability (e.g., *affected version*, *vulnerable function*, *system call*, etc.) from the CVE description and the corresponding Linux git log, which enables vulnerability call-chain identification (②). Then, DiffCVSS compiles the Linux kernel (with *affected version*) and determines vulnerability call-chains using artifacts (e.g., vulnerability-related functions and tokens) extracted in previous step. Such information is further used to identify and match the corresponding vulnerability call-chains in the affected Android kernel (③). Given both Linux and Android vulnerability call-chains, DiffCVSS conducts a differential analysis to identify the vulnerability path differences (functions), and further determine how such difference will affect vulnerability severity level, by examining the function in the call-chains and their associated CVSS metrics (④).

4.3 Design

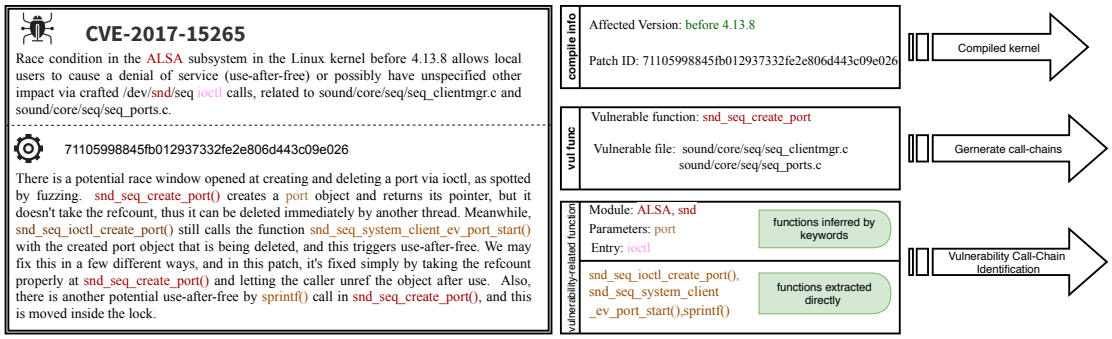

Figure 4.2: CVE description and Linux git log of CVE-2017-15265.

In this section, we will detail the design of DiffCVSS.

4.3.1 Mapping Metrics to Functions

As the first component, DiffCVSS maps exploitability metrics to functions: (1) identifying functions that are related to the CVSS metrics and (2) mapping the CVSS metrics value to the functions. We decide to perform the function-based mapping for two reasons. First, we found that, in most cases, the severity assessment determines metric values at a granularity of functions. Second, function description in source code provides a direct and easy way for developers to understand the functionality, parameters, or the usages of a function. Hence, we can use NLP techniques to automatically analyze those descriptions to identify functions that are related to the CVSS metrics and to construct the mapping.

For example, the Linux function tcp_rcv_established has the description of *"TCP receive function for the ESTABLISHED state"*. This description indicates the function is bound to the network stack (i.e., `AV:N`). We can thus construct a mapping between tcp_rcv_established and `AV:N`. We elaborate on our design as follows.

Function-description extraction. To extract function descriptions, we first use *Sphinx* [154] to automatically identify well-structured descriptions in the kernel-doc format from kernel source files. However, less-structured descriptions are common (around 67.6%) that cannot be directly extracted by *Sphinx*. To address this, we use regular expression to extract them. Specifically, we first use Coccinelle [155], a tool for pattern matching and text transformation, to extract the function name and its line number from source code. Then, we design regex expressions to capture single-line and multi-line block descriptions above the function. As a quick evaluation, we manually sample 200 functions with less-structured descriptions for testing. The results show our regex-based method is very effective—achieving a recall of 100% and a precision of 99.5%. As a result, we gather 48,232 function descriptions by using *Sphinx* and 100,778 more using the regular expressions, which can cover all of core kernel functions [156].

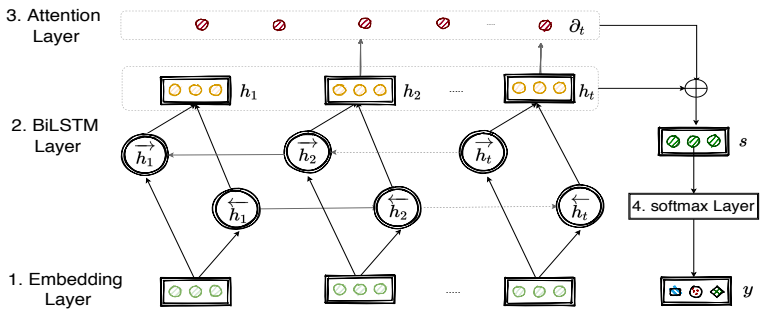

Figure 4.3: BiLSTM+Attention model

Inferring CVSS metrics for functions. After gathering function descriptions, DiffCVSS then infers CVSS metrics for each function based on their descriptions. In our study, we use BiLSTM [141] and attention mechanism [157] for function-description reasoning and exploitability-metric classification. We choose such a model for two reasons. First, some descriptions are relatively long (more than 100 words), hence we use the BiLSTM model which is able to memorize longer sequences of the input data. Second, after manually reviewing hundreds of ground-truth data, we found some informative keywords that have decisive impact on the functionality of functions, which can be captured by the attention mechanism. For example, if a function description has the words such as "permission", "privilege", "admin", "capability", it has a high chance to be associated with PR:H. Note that those informative words are learned by the self-attention mechanism instead of manually observation. In particular, our BiLSTM consists of two LSTM units, which operate in both directions to capture long-term dependencies between word sequences. Also, the attention mechanism can automatically focus on the words that have a decisive effect on the classification to capture the most critical sentimental information in a sentence.

More specifically, we first represent sentences in the descriptions into vectors. We concatenate each word's vector generated by words embedding [158]. Based on this, DiffCVSS further uses BiLSTM [141] and the attention mechanism [157] to discover metric-related functions. As shown in Figure 4.3, our model consists of four components: (1) Input layer which is the sentence vector $emb\_s = \{e\_1, e\_2, ...e\_T\}$, concatenated by the each token's vectors e_i that is output by the pre-trained Word2vec's skip-gram model. (2) LSTM layer which contains two sub-networks to learn left and right sequence

contexts respectively. The outputs are the word annotations $h\_i = \{\overrightarrow{h\_i} \oplus \overleftarrow{h\_i}\}$, where $\oplus$ is the concatenate operation. (3) Attention layer: considering that not all context words have the equal contribution to the semantics of a sentence, we use a self-attention layer to automatically capture important parts of the sentence itself. The output of attention layers is $s = \sum \_t\partial\_th\_t$, where ∂ is an attention weight, s is the output sentence vector, and t is the word sequence. (4) Output layer: further, s is the input to the softmax layer for exploitability-metric classification, i.e., $y = \text{soft}\max(W\_ss + b\_s)$, where W_s is the weighted matrix, and b_s is the bias.

Note that each function can be associated with more than one exploitability metrics. For example, file_ns_capable can be mapped to two metrics PR:H and UI:R, because it is a file operation that needs user interaction while it is also a permission check that determines if the operator of that file has a permission. Hence, in our study, we train a classifier for each exploitability metric (see §4.4).

Transforming model to Android domain. In order to avoid excessive human work in labeling functions in Android kernels, we transform the semantic knowledge learned from Linux kernel to Android. Our key insights are two-fold. First, an Android kernel is built on top of the Linux kernel, and they share around 84% of functionalities [159]. Second, although the Android kernel introduces various Android-specific facilities, such as ashmem (Android shared memory driver), Binder IPC mechanism, and wake lock mechanisms [160], the informative words that have significant impacts on the classification results should share the same or similar meaning. For example, the function sdcardfs_permission is an Android-specific function, used to perform permission check on the sdcardfs inode. Its description is *"calling process should have AID_SDCARD_RW permission"*. Although AID_SDCARD_RW is an Android-specific term, the informative keyword "permission" here is inline with the Linux kernel.

In order to keep such similarity and mitigate the subtle platform differences, we fine-tune the model transferred from the Linux kernel using a small number of data which are specific to the Android domain. Particularly, we freeze attention layers to preserve learned informative keywords and at same time adjust hidden-layers using such fine-tuning data to make the transferred model optimized for the Android kernel. More specifically, DiffCVSS copies the parameters in the attention layers from the Linux kernel model to the Android kernel domain. Then, DiffCVSS fine-tunes the transferred

model based on the data selected in §4.5.1. DiffCVSS will enumerate all the different combinations of the hyper-parameters and choose the one with the best performance. Those hyper-parameters include different optimizers, dropout for regularization, learning rate, and epoch.

Discussion. To evaluate the reliability of function descriptions on assessing the severity of vulnerabilities, we manually investigate a ground truth dataset (see §4.5.1) and find that the function descriptions can effectively indicate the severity of vulnerabilities. Specifically, we manually look into all the vulnerability call-chains recorded in the fuzzing log and find 489 functions with descriptions that directly reflect the exploitability metrics values. For example, the description of function tomoyo_check_unix_acl is *"Check permission for Unix domain socket operation"*, which provides highly relevant information about exploitability metrics PR. Also, for all the ground-truth vulnerabilities, on average, 85.1% of their exploitability metrics can be directly reflected in the descriptions of functions on the vulnerability call-chain. The result shows that most vulnerability call-chains contain enough functions that have severity-related descriptions.

4.3.2 Vulnerability Artifact Recognition

As mentioned earlier, semantic information (including affected version, vulnerable functions, and system calls) of vulnerability paths comes from the text content of CVE and Linux git log. In our study, to rebuild the vulnerability path given a vulnerability, we will retrieve (1) compilation-related information (i.e., affected version, configuration options), which provides settings for us to compile kernel into LLVM IR. (2) vulnerability entry points and endpoints (i.e., system call, vulnerable function), which enable us to generate possible vulnerability call-chain in the call graph. (3) vulnerability-related functions and tokens (e.g., module name, macro name), which helps us determine functions (except for vulnerable function) in the vulnerability path.

Retrieving affected version, vulnerable function, and system call. We adopt the method used in Semfuzz [93], which uses both regex expression and constituency tree that represent the syntactic structure of a sentence [161], to recognize affected version, vulnerable function, and system call in the CVE and Linux git log.

Identifying affected configuration. The configuration information indicates whether

driver is built into the kernel (e.g., CONFIG_XFRM_MIGRSTE=y) or is not selected (e.g., CONFIG_XFRM_MIGRSTE=n). Those options are specified in the config file of the kernel, e.g., .config in the Linux. To retrieve those information, we use regex "`\bCONFIG\_\w+`" to identify the configuration name and operation. After that, we use its semantic context to determine its option ("y" or "n"). Specifically, we construct a dependency tree using spaCy [162] to identify the verb of configuration name. If the verb is either "enable", "use", "enforce" or "build" and there is no negation modifier before the verb, we will regard the configuration option is "y". However, if the verb contains negative meaning (e.g., "disable") or there is an negation modifier dominating the verb (e.g., "is not enabled"), we will view configuration option is "n". In our study, we use spaCy [162] to identify negation modifiers.

Recognizing and inferring vulnerability-related functions. Here we defined vulnerability-related functions as the functions in the vulnerability path. Such inferred functions will facilitate the identification of vulnerability call-chains (see §4.3.3). To this end, we generate a list of Linux function names using Coccinelle [155] and match those functions names in the text. However, not all vulnerability-related functions are recorded in the CVE or git log, but only some keywords (e.g., ioctl, which can be correlated to the functions do_vfs_ioctl, vfs_ioctl). Hence, in our study, we retrieve those keywords (i.e., vulnerability-related tokens) and further infer functions associated with those keywords. More specifically, after manually examining 100 CVE and git logs, we determine three kinds of vulnerability-related tokens: module name, variable name/type, macro name. After that, we generate the list of all module names, variables, macro names in the Linux kernel by building parsers on top of Coccinelle [155]. In this way, we achieve three lists with 2,538 module names, 81,327 variable name/type, 1,903,662 macro names.

Given those lists and associated types, we retrieve vulnerability-related tokens in the CVE and git log, by considering the semantic context of those tokens instead of the simple approach (string matching) which failed to consider the grammatical property of the words in sentence. For example, trigger acts as a variable in the function static void save_ELCR(char* trigger). However, in CVE description or git log, "trigger" is usually used as a verb (e.g., "to trigger buffer overflow"). Specifically, DiffCVSS uses Part-of-Speech (POS) tagger in spaCy [162] to recognize the grammatical property (e.g., noun, verb, adjective) of each word. If the token appears in the parse tree and its POS

tag is either NOUN or PROPN or ADJ [162], we regard it as a vulnerability-related token. Such approach yields an accuracy of 96% to recognize vulnerability-related tokens in the CVE and git log. After that, we correlate such tokens to functions by checking if they appear in a function's description or function name or function body.

4.3.3 Vulnerability Call-Chain Identification

As we discussed in §4.1.3, to assess exploitability metrics of a vulnerability, we need to know its vulnerability call-chains (from entry points to the vulnerable function). Instead of using the symbolic execution or directed fuzzing, which suffers from scalability and coverage issues, DiffCVSS leverages vulnerability information to automatically identify the vulnerability call-chains in the Linux and the Android kernel. As will be shown in §4.6, such an approach is not only scalable but also precise.

Table 4.3: Mapping CVE keywords to functions in call chains.

Some funcs in the selected call-chain	Related keywords
snd_seq_create_port	snd
snd_seq_ioctl_create_port	ioctl, snd
snd_seq_ioctl	ioctl, snd
vfs_ioctl	ioctl
do_vfs_ioctl	ioctl
SYSC_ioctl	ioctl
SyS_ioctl	ioctl

Identifying vulnerability call-chains in Linux. To get the vulnerability call-chains in the Linux kernel, DiffCVSS first leverages the vulnerability patches and CVE description to find all the related functions, and applies two rules to identify a call-chain as the vulnerability call-chain if (1) it contains a highest number of related functions; and (2) the functions in the call-chain should also match with the same severity metrics specified in the CVSS. Taking CVE-2017-15265 as an example, given its vulnerable function snd_seq_create_port, DiffCVSS identifies 514 call chains from different entry points. However, based on the description in Figure 4.2, DiffCVSS will identify several keywords, such as sound, snd, and ALSA. By using these keywords to find the related functions (see Table 4.3) and match with them, DiffCVSS can uniquely identify the vulnerability call-chain.

Matching vulnerability call-chains in Android. Since the CVSS is evaluated for the Linux kernel instead of Android, we cannot directly use the CVE description

to identify the corresponding vulnerability call-chain in the Android. To address this, we propose a method to match the most relevant vulnerability call-chain in Android based on a fact that most functions are still the same or similar between the two kernels. We use the following formula to evaluate the similarity between two call-chains (one in Linux and the other in Android). The idea is quite simple and intuitive—we perform a similarity analysis against the two call-chains, and the similarity is defined based on two intuitions: (1) similar call-chains should call many same functions in the same order, and (2) the shared functions should also be similarly distributed in the call chains. That is, they also share a similar structure.

Accordingly, we define the similarity as, $Sim = std(index(LCS(CC_L, CC_A))) * len(LCS(CC_L, CC_A))$, where CC_L and CC_A are the call-chains in Linux and Android, respectively; LCS is the longest common subsequence, which is commonly used to measure the edit distance between two lists and used in previous works such as [163], to measure the similarity of call-chains; $index()$ is to get the indexes of shared items in the Linux call chain and LCS; std is the standard deviation(std) of the indexes list. std is a measure of dispersion for the shared functions; a higher std indicates that the shared functions are spread out over a broader range of the call chain. Thus, the higher the Sim is, the more similar the two call-chains are.

Addressing the path-explosion problem. It is unrealistic to explore all call-chains due to the path-explosion problem. We observe that 95% of feasible paths collected from the fuzzing log generated by Syzkaller [47] contain less than 18 functions. Based on this observation, we employ the Dijkstra's algorithm [164] (a algorithm for obtaining the shortest paths between two nodes in a graph) to select paths with less than 18 functions. Our evaluation results in §4.6 show that this approach only introduces about 4/65 (6%) of false negatives. However, without such a limit, there will be almost an "infinite" number of reachable paths from an entry point to a vulnerable function—the complexity is O(V!) [165], where V is the number of vertices in the call graph; we found that the V is larger than 300K in the recent versions of the Linux kernel, easily leading to path explosion. Therefore, we believe that choosing such a limit of 18 functions is necessary.

4.3.4 Differential Severity Analysis

After identifying and matching the vulnerability call-chains in Linux and Android, DiffCVSS analyzes their severity differences. DiffCVSS first uses the function-metric mapper (§4.3.1) to determine whether the functions in the Android vulnerability call-chain are associated with exploitability metric values, based on which DiffCVSS can inference the values for each exploitability metric. Notice that, for a specific metric, if multiple values are found in the call-chain, DiffCVSS will choose the value associated with higher exploit requirements. For example, if DiffCVSS finds two different functions in the Android vulnerability call-chain, one is associated with AV:N and the other is associated with AV:P, the final value for exploitability metric AV will be P. This is because the attacker has to access the vulnerable machine physically (AV:P), which is a higher exploit requirement than remotely accessing the machine (AV:N). After that, DiffCVSS employs differential analysis to compare the exploitability metrics in the Android and with the original CVSS vectors in the CVE database. In this way, DiffCVSS outputs the differential metric values for the vulnerability in Linux and Android.

For instance, given a cross-OS vulnerability CVE-2016-2085 with the function inode_permission in the differential call-chains, DiffCVSS will map such a function to the metric value PR:H. When comparing with the original metric value of PR:N, we conclude that an attacker requires higher privileges when exploiting the vulnerable component.

Metrics-severity rating and comparison. Given those differential metric values in Linux and Android, DiffCVSS quantifies severity changes using the CVSS calculator [10], by mapping those metric values into real numbers. Note that the quantification focuses on only the differential metrics, which is a limited number, so it is easily automatable. Using the same example of the vulnerability CVE-2016-2085, which differential metrics are PR: H, and AC: H; after calculating the severity changes, the results show that this vulnerability has a lower severity in Android than Linux.

4.4 Implementation

Word2Vec model training. We train the Word2vec model using gensim [166]. The size of word vector is 300 (the commonly-used value); the window size is 5 (maximum distance

between current words and predicted words); and min_count is set to 1 (consider all the words appear in the corpus). The training corpora includes 149k function descriptions from the Linux kernel, 145K function descriptions from the Android kernel, 3k Linux-related CVE descriptions, and 935K git log messages. We pre-process each text sequence by removing white space and stopwords, transforming hump-expressed or underline-expressed function names into separate words (e.g., check_ipc_perms -> check ipc perms), expanding constructions (don't -> do not), etc.

BiLSTM+Attention model training. We manually annotated 5,594 functions in total for model training. Using the aforementioned model architecture (§4.3.1), we train a multi-classifier for AV and three binary-classifiers for AC, PR, and UI, respectively. We implement our models using *Tensorflow* [167]. The embedding size is set to 300 (same as the word2vec). The hidden size used in BiLSTM is 150. The attention layer is initialized with normal distribution. The dropout rate is 0.2. In the dense layer, we use the *softmax* as the activation function. Also, we use the categorical cross-entropy loss and Adam optimizer with learning rate 0.0001 in the model training.

OS-kernel compilation. In order to compile the target kernel given a vulnerability and its CVE description, we first leverage the information extracted in §4.3.2 to determine configuration options and the architecture. For example, if the vulnerability can only be exploited when CONFIG_XFRM_MIGRSTE is disabled in X86 module, we will set CONFIG_XFRM_MIGRSTE=n in the config file and set ARCH=x86 in make options. If there is no such information extracted for CVE description or git message, by default, we will use the *allyes* configuration in the *aarch64* architecture. Specifically, for the Linux kernel compilation, we use standard Clang to generate bitcode files. For Android compilation, we use AOSP Clang which provides pre-built tool chains in different architectures. However, the process becomes tedious when some kernel versions do not support the compilation with Clang (e.g., version before 4.4.165 or 4.9.139). To address this, we back-ported the Clang patch-set before compiling it.

Building call graph and call chain. To identify call-chains in different systems, we first build call graphs for each of them. Specifically, we analyze all the call instructions based on LLVM and leverage the state-of-the-art type matching [36, 51, 168] to handle indirect calls. Furthermore, based on the call stack and the call-graph, DiffCVSS leverages flow-sensitive analysis to build the call-chain by inserting the called functions into the

call stack.

4.5 Evaluation

Table 4.4: The groundtruth set of mapping functions to metrics.

Metrics type	Metrics value	CVE	APIs	Example
Attack Vector	*N*	22	34	*tcp_rcv_established*: TCP receive function for the ESTABLISHED state
	A	1	2	*wlan_setup*: set up any members of the wlan device structure that are common to all devices
	P	24	116	*device_release_driver*: Manually detach device from driver. When called for a USB interface
Attack Complexity	*H*	27	32	*drm_atomic_check_only*: check whether a given config would work
Privileges Required	*H*	5	6	*tomoyo_check_unix_acl*: Check permission for unix domain socket operation
User Interaction	*R*	22	42	*tiocgsid: @tty: tty passed by user, @real_tty*: tty side of the tty passed by the user if a pty else the tty

4.5.1 Experiment Setting

Platform. We use a set of computing resources available to us, including two servers (96 cores/256GB memory, 12 cores/64GB memory, respectively), and two desktops (8 cores/64GB memor/2 GPUs for each of them). All these machines are running on Ubuntu 20.04.

Dataset. To evaluate the effectiveness of DiffCVSS, we utilized the following datasets.

• *Ground-truth dataset for mapping functions to metrics.* Our tool *api2Metrics* mapped functions to CVSS metrics based on attention-based classifiers. In order to train the models and test their performance, we create a ground-truth dataset, which has been released at [169]. The labeling process is as follows. We first collect vulnerabilities that contain fuzzing logs and extract their corresponding CVSS metrics assigned by NVD. As shown in Table 4.4, we found 22 vulnerabilities with UI:R; 22 vulnerabilities with AV:N; 24 vulnerabilities with AV:P; 1 vulnerability with AV:A; 27 vulnerabilities with AC:H; 5 vulnerabilities with PR:H. Then, two annotators with security background manually check functions in the fuzzing logs, map them into related metrics. In total, we collected 152 functions in AV metric, 32 functions in AC metric, 6 functions in PR metric, and 42 functions in UI. Such data serve as a good guidance for us to label more data. Two annotators further labeled 1,557 functions for AV metric, 1,529 functions for AC metric, 1,371 functions for PR, and 1,137 functions for UI. Finally, we integrate all labeled functions. On average, we have an agreement rate as 95%. For those uncertain cases, we contact NVD maintainers for answers. For example, the function btrfs_read_fs_root (a

file operation) appears in the fuzzing log of CVE-2019-19036. We are not sure whether it should be associated with UI metric. The response from NVD shows that when a CVE requires a file to be executed in order to exploit, the UI should be R. Hence, we label such file operations as UI related.

• *Ground-truth dataset for mapping functions to metrics in different versions of Android.* As our metric mapping tool is trained on the labeled functions from Linux mainline, we need to transform it into Android. We build a ground-truth data set from three stable Android versions which are Android-3.18-o-release, Android-4.19-q-release, Android-12-5.4. As demonstrated by prior work [160], the Android kernel introduces a number of new kernel subsystems and new mechanisms. Take Android-4.14 as an instance, the largest features changed from mainline include 13.8% in Networking (net/netfilter), 13.5% in Sdcardfs (fs/sdcardfs), 9.4% in USB (drivers/usb), and so on. In order to better migrate the difference, we label data from such android-enhanced functions. More specifically, we first identify those Android-specific functions which only appear in Android kernel. In total, we get 22,169 Android-specific functions in Android-3.18-o-release, 8,695 in Android-4.14, and 4,079 in Android-12-5.4. Further, we label 150 functions for each metric of each version as our ground-truth.

• *Ground-truth dataset for vulnerability call-chains.* To evaluate the vulnerability call-chain identification of DiffCVSS, we collect 65 vulnerabilities in CVE database, which have recorded the fuzzing logs, including the call-chains from entry functions to vulnerable functions. We use these vulnerabilities and the associated call-chains as the ground-truth set in this evaluation.

4.5.2 Evaluating Metric-to-Function Mappings

In a nutshell, we achieve a high accuracy in mapping metrics to functions: a precision of 93.0% and a recall of 91% on average. In this study, we perform a Train-Test Split of our labeled data. Specifically, we randomly sample 70% of data to train the model, 10% of data to tune the hyperparameters, and the rest 20% to evaluate the model performance. Table 4.5 details the experiment results.

4.5.3 Precision and Recall of Classifiers

Attack Vector (AV) classifier. To train this multi-class classification model, we manually label 1,557 functions. Based on the rules provided by CVSS [133], 190 functions are bounded to network stack and allow remotely access (N); 124 functions are also bounded to network stack but limit network attacks to adjacent access (A); 203 functions require attackers' physically access (P); the remaining 1,101 functions are not related to AV metrics. The results are shown in Table 4.5. When looking into the false positives cases of N and false negatives of A, we found that the classifier falsely classified some adjacent network functions into N. This is due to they share the similar semantic contexts, as the metric values N and A are both bound to network stack; the difference is that metric value A can only be locally accessed (e.g., Bluetooth or IEEE 802.11) while N can be remotely (e.g., across one or more routers). In our study, our attention mechanism is able to capture some informative keywords which indicate the same shared physical network or local network (e.g., "WLAN", "wireless", "Bluetooth", "wifi", "ieee80211") to distinguish A from N.

Attack Complexity (AC) classifier. We train a binary classifier to discover functions that reflect complex conditions that attacker must control to exploit the vulnerability. For this purpose, we manually label 1,529 functions, among which 411 functions reflect high attack complexity (H). As shown in Table 4.5, on the test data, we achieved 92.38% precision and 91.51% recall in classifying high attack complexity functions. When analyzing the false positives of the model, we found that the falsely labeled functions turn out to indeed contain sentiment terms and reflect high requirements for exploitability, whose semantic context is more focused on the requirement of access privileges that are supposed to be classified as PR metric. For example, the sentence *"check for access right to given inode."* are falsely labeled, since it includes the sentiment word "check" and describe the need for extra capability. However, the corresponding function inode_permission is intended to check the read and write permission on an inode which should be classified into a separate PR metric according to the latest CVSS 3.1 guideline. On the other side, false negatives are mainly caused by the sentiment analysis, which failed to put more attention to some sentiment terms like "futex" which implement basic locking and indicate the timing conditions, due to the incompleteness of training set.

Privilege Required (PR) classifier. We train a binary classifier to discover functions that reflect certain permission is required to perform attack. To this end, we manually label 1,371 functions, among which 236 functions perform permission checks. On the test dataset, our model achieves a recall of 94.52% and a precision of 93.24%. When looking into false positives, we found that those falsely labeled functions indicate some other conditions the attacker needs to control, which however actually belong to the AC metric. For example, the function qla4_82xx_pci_mem_bound_check has the description *"check memory access boundary used by test agent support ddr access only for now"*, which however indicates more conditions the attacker should control during exploitation and hence is supposed to be classified as AC. Interestingly, such blurs between AC and PR metric is explainable by historical CVSS version (2.0), in which AC and PR both belong to the same metric Access Complexity[170]. When looking to the false negatives, we found many of them are caused by less formal, imprecise, vague descriptions [171].

User Interaction (UI) classifier. We train a binary classifier to recognize functions that reflect user operations. For this purpose, we manually label 1,137 functions. On the test dataset, our model achieves a precision of 92.96% and a recall of 91.67%. When looking into the false positives, we found that the falsely labeled functions are caused by high attention to some specific terms. For example, the function account_user_time has description *"account user cpu time to process the process that the cpu time get accounted to cputime the cpu time spent in user space since the last update"*, which is falsely labeled as the excessive attention to the informative term "user".

4.5.4 Model Transferability

In order to evaluate the model's transferability on Android kernel, we ran the four classifiers over the ground-truth dataset which contains labeled functions from three stable Android kernel versions. As shown in Table Table 4.5, the performance on Android is in parallel with that of Linux, which confirms the stability and generality of our models. For example, when classifying the functions to PR: H, the model achieves a recall around 93% in both Android and Linux kernels.

Table 4.5: The precision and recall of each classifier on multiple Linux and Android versions. M = Metrics.

		Linux Mainline		Android-3.18-o-release		Android-4.19-q-release		Android-12-5.4	
Metrics	Label	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Attack Vector	N	92.42%	93.84%	94.44%	87.2%	89.74%	92.11%	94.45%	91.23%
	A	87.50%	93.33%	86.04%	92.5%	93.33%	90.32%	91.11%	93.18%
	P	88.52%	91.53%	89.66%	92.85%	91.43%	94.12%	87.88%	93.55%
Attack Complexity	H	91.51%	92.38%	93.88%	86.79%	93.44%	89.1%	93.1%	91.53%
Privileges Required	H	94.52%	93.24%	93.94%	91.8%	92.59%	89.28%	91.67%	91.67%
User Interaction	R	91.67%	92.96%	93.65%	90.77%	92.96%	90.04%	92.5%	92.72%

Table 4.6: The precision and recall of each classifier on multiple Linux and Android versions. M = Metrics.

		Linux-4-4		Linux-4-9		Linux-4-14		Android-4.4-0		Android-4.9-p		Android-4.14-q	
M	Label	R	P	R	P	R	P	R	P	R	P	R	P
AV	N	94.59%	89.74%	90.14%	91.42%	92.85%	91.54%	92%	89.61%	88.24%	91.83%	91.17%	87.32%
	A	89.47%	91.07%	90.14%	91.42%	92.85%	91.54%	90%	93.10%	92.15%	90.38%	92.92%	92.10%
	P	92.45%	89.09%	89.65%	92.85%	90.91%	89.28%	88.23%	93.75%	90.56%	85.71%	87.80%	90%
AC	H	90.74%	92.45%	89.19%	91.66%	90.52%	92.47%	90.74%	89.91%	92.92%	92.11%	92%	90.19%
PR	H	90.14%	92.75%	92.55%	93.54%	93.54%	89.23%	89.83%	93.81%	94.04%	89.77%	93.90%	90.58%
UI	R	91.01%	94.18%	92.41%	90.12%	91.57%	92.68%	89.87%	91.03%	92.71%	90.81%	90.91%	93.33%

4.5.5 Effectiveness on Different Versions

This section further evaluates the models of DiffCVSS against more versions of the Linux and Android kernels. The evaluation is to confirm that DiffCVSS is generic and has stable performance across different versions. Specifically, we evaluate the performance of DiffCVSS on Linux-4.4, Linux-4.9, Linux-4.14, Android-4.4-o, Android-4.9.p, Android-4.14-q. We randomly sample and annotate 200 distinct functions for each metric under each Linux and Android version. Further, we run Linux and Android kernel models, respectively, and the results are detailed in Table 4.6. As we can see, the precision and recall of each metric over different versions are numerically stable. For example, the precision of the PR metric across three Android kernel versions is 91.39% on average with the standard deviation of 1.74. Moreover, when we inspect the internal function difference in three Linux versions and three Android versions, we found that the function difference is negligible, and most of the functions would not be changed between different versions. Specifically, for two adjacent versions listed above, such as v4.4 and v4.9, on average, the newer version will add 9.8% of functions and delete about 3.9% of functions in the old version. Such observation explains why our Linux model has a stable performance across different versions, the same as the Android model.

4.5.6 Comparison with the State of the Art

Pex [168] is a recent tool that identifies a set of functions that perform permission checks. More specifically, Pex manually constructed a small set of known permission-check functions, and then used dominator analysis [172] to find their wrappers. In total, PeX finds 284 functions that perform permission checks. DiffCVSS is able to map all of them to the metric value of PR:H. Moreover, DiffCVSS discovers additional 1,034 permission-check functions through the Privilege Required classifier.

4.6 Evaluating Call-Chain Identification

The scale of possible call-chains. Given a vulnerability, DiffCVSS first collects all possible call-chains and then identifies the one related to the vulnerability. If there are too many possible call-chains, the identification may not scale. The evaluation shows that, on average, DiffCVSS collects 352 possible call-chains. With the call-chain identification mechanism, DiffCVSS is able to precisely identify 7.7 vulnerability call-chains on average (with the median of 2). This result shows that DiffCVSS can effectively mark 98% of call-chains as irrelevant.

Effectiveness of vulnerability call-chain identification. As discussed in §4.5.1, we selected 65 vulnerabilities with fuzzing log as the ground-truth set to evaluate the precision of our approach. Our evaluation result shows that 54 (83%) of these vulnerability call-chains can be identified by DiffCVSS, and 11 of them are missed due to the following reasons. First, the inaccuracy of call-graph construction. In the Linux kernel, some entry functions are written in assembly code, which cannot be correctly compiled and analyzed by LLVM. Therefore, the callee of these entry functions may be missed. This leads to 7 missed cases. Also, as we discussed in §4.3.3, DiffCVSS enforces a limit of 18 for the number of functions in a call-chain to avoid path explosion. This leads to the remaining 4 missed cases. Accordingly, these issues can be alleviated in the future by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. However, improving such techniques is challenging, which requires new designs and lots of engineering works, and thus they are regarded as the future works for DiffCVSS.

Precision of the Android and Linux call-chain matching. As we discussed in §4.3.3, by comparing the CVE-related call-chain in Linux, DiffCVSS matches the most similar call-chain in Android and further analyzes the metrics of this call-chain. Here we

evaluate the precision of the matching. We manually compare the Linux vulnerability call-chain with Android call-chains identified by DiffCVSS to see if they contain the same set of functions in the same execution order. It took 2 security professionals 2 person-hours for data annotation.

After checking all the 127 call-chain pairs, we found that 113 of them are matched exactly, but 14 are not exactly the same. We further analyzed these 14 cases and found that all of them are not caused by the similarity analysis, but instead caused by missing the same functions in Android. This means that these 14 cases may not be false-positive cases, but are already the most similar call chains we can find. Therefore, given a CVE-related call-chain in Linux, we believe that the similarity analysis is precise in capturing the similar Android call-chain based on this result.

4.6.1 Evaluating Cross-OS Severity Differences

In this section, we evaluate the severity differences of cross-OS vulnerabilities in Linux and Android, as well as in different versions of Linux.

Table 4.7: Cross-OS vulnerability exploitability metric difference between Linux and Android.

	# **vulnerabilities**			
	AV	**AC**	**PR**	**UI**
More severe in Android	13	11	35	2
More severe in Linux	63	57	36	75
Similar severe in Linux and Android	51	59	56	50

4.6.2 Severity Differences Between Linux and Android

To conduct this experiment, we select cross-OS vulnerabilities from the Linux kernel with the following rules: (1) the vulnerabilities should be found in recent years because as cross-OS vulnerabilities, they should affect at least one of the versions in Android (v3.18, v4.4, v4.9, v4.14, v4.19, and v5.4); (2) the patch of the vulnerability is available; (3) the vulnerable file can be successfully compiled to LLVM IR. Finally, 127 vulnerabilities are selected and analyzed in this experiment.

As discussed in §4.3.4, based on the differential severity analysis, DiffCVSS outputs differential CVSS metric values of cross-OS vulnerabilities in Linux and Android. The results are summarized in Table 4.7. For example, the first row in the table indicates the

number of CVEs that the corresponding metrics have higher severity than they are in the Linux (e.g., a vulnerability has AV:N in Android but AV:P in Linux). Furthermore, our study also measures the difference of the vulnerability's severity groups (low, medium, high, critical), defined by CVSS [173]. This result shows that among 127 vulnerabilities, beyond 17 (13%) vulnerabilities with the same severity in Android and Linux, the severity of most of the cross-OS vulnerabilities (87%) is different in different OSes. Specifically, 92 (72%) of the vulnerabilities are more severe in Linux than Android, which means that prioritizing the patch for Android may waste maintenance resources that are supposed to be allocated for critical vulnerabilities. Also, 18 (15%) vulnerabilities are more severe in Android, which means that these vulnerabilities may not be patched timely in Android if the severity evaluation is based on the CVSS score for Linux. This can be particularly critical as adversaries will have a larger time window to exploit the "already-publicized" critical vulnerabilities in Android devices.

The precision. To check the precision, we manually look into all these cases reported by DiffCVSS and see if (1) the identified exploitable call chain is indeed related to the CVE description, (2) the identified exploitability metrics from functions are correct, and (3) the severity differences are correctly calculated and compared. If a case meets all of these requirements, we regard it as correct. The manual analysis shows that among these 127 cases, 116 of them are correct, which means that DiffCVSS achieves a high precision of (91.3%) in the differential severity analysis. Looking into these incorrect cases, 4 are caused by missing enough useful vulnerability artifacts to select the vulnerability call-chains. Therefore, DiffCVSS mis-selected the vulnerability call-chains. Also, 7 are caused by the incorrect mapping from exploitability metrics to functions. This result is aligned with the result from the user study (see §4.7), and we will further discuss the potential improvements for precision in §4.8.

Case Study: a more severe vulnerability in Android. CVE-2019-3701 is a local out-of-bound write vulnerability. Its exploitability metrics assigned by NVD in the affected Linux kernel v4.19 are AV:L/AC:L/PR:H/UI:N. Running on this vulnerability, DiffCVSS outputs a differential metric value of AV:N in affected Android kernel v4.19. It indicates that an attacker can even exploit this vulnerability remotely in Android (AV:N)—a much more severe case—while in the Linux kernel, an attacker has to access the target system locally (AV:L). When looking into the difference of vulnerability call-chains

in Linux and Android, we found that the function tcp_v6_do_rcv exists in Android vulnerability call-chain while not in Linux. The function tcp_v6_do_rcv is the network protocol-level related function, which indicates the vulnerable component is bound to the network stack in Android.

4.6.3 Severity Differences Between Linux Versions

In this evaluation, we further test the vulnerabilities-severity differences among different versions of the Linux kernel. Since there are thousands of versions of Linux kernels, testing all of them is unrealistic. Therefore, in this evaluation, we only test the vulnerabilities-severity differences in several commonly-used versions and long time support versions, including v3.8, v3.18, v4.4, v4.9, v4.19, v5.1, and v5.4. Then, from all the 127 vulnerabilities we have tested, we select the vulnerabilities that would affect at least two of the versions we just mentioned. Finally, 92 vulnerabilities are selected and analyzed. Among them, 62 vulnerabilities have the same severity across different versions of the Linux kernel, and 30 show different severity in different versions. These results indicate that even for the same system, vulnerabilities can cause different severity for different versions. Therefore, the patching priority should also be evaluated per version when needed.

4.7 Usability Study

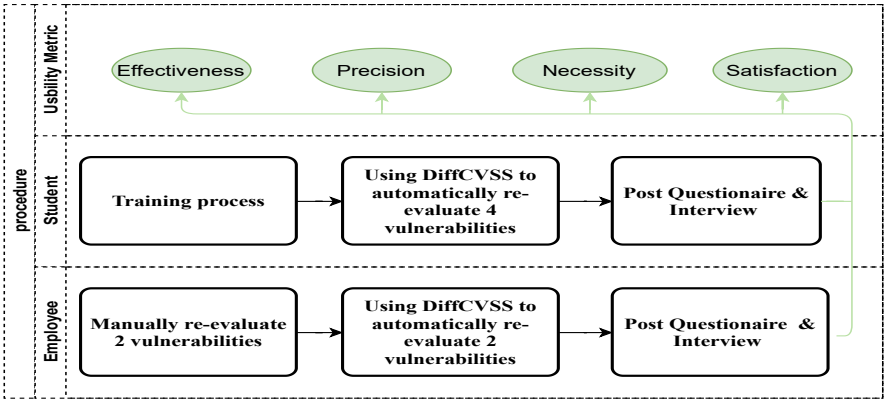

Figure 4.4: The Procedure of the user study.

We further conduct a user study to evaluate the usability of DiffCVSS from the user perspective (e.g., downstream maintaixfners). The usability of DiffCVSS focuses on effectiveness, accuracy, and satisfaction. In particular, we seek to answer the following key questions: **Q1:** How efficient is DiffCVSS in reducing maintainer workload? **Q2:** How accurate is DiffCVSS in re-evaluating vulnerability? **Q3:** How usable is DiffCVSS in practice?

Recruitment. After an IRB approval, we recruited participants by distributing recruitment advertisements online (see the detail requirements for recruitment in Appendix A), contacting related organizations (mostly CVE Numbering Authorities (CNAs) worldwide) that maintain downstream Linux derivatives, and snowball sampling, where participants recommended other colleagues. In total, we recruited 30 participants, including ten maintainers in industry who have real-world experience in vulnerability evaluation and 20 graduate students who have a background in system security. We follow a standard and ethical way [174, 175, 176] to reward participants ($30 Amazon gift card for each student and $100 for each employee) in the user study. Table 4.8 details the demographics of participants. We believe the number of participants is substantial, as it is already more than 12-20 participants as suggested by qualitative research best practices literature [177] and also aligns with related works [178, 179, 180, 181].

		S (n = 30)
Organization	U1	40%
	U2	35%
	U3	20%
	U4	5%
	C1	40%
	C2	20%
	C3	20%
	C4	10%
	C5	5%
	C6	5%
years	1-3 years (Security analyzer)	30%
and role	3-5 years (Security analyzer)	50%
	6+ years (Security Manager)	20%
Education	Bachelor's degree	10%
	Master's degree	20%
	Ph.D's degree or higher	70%

Table 4.8: PARTICIPANTS DEMOGRAPHICS: U1-U4 represents 4 universities,C1-C6 represents 6 companies, S represents Survey.

Procedure of the user study. In this study, we selected in total 20 vulnerabilities analyzed by DiffCVSS, which can cover different exploitability metrics and different severity levels, and every participant is required to analyze 4 of these vulnerabilities. Due to the various expert levels of maintainers and students, the procedures slightly differ, which are shown in Figure 4.4. Specifically, maintainers were asked to re-evaluate two vulnerabilities manually and the other two with the help of our tool, DiffCVSS. To ensure students are capable for vulnerability assessment, we asked them to study the training materials before re-evaluating the four vulnerabilities with the help of DiffCVSS. Our training materials include wiki-based background introduction, real-world examples of vulnerability-severity re-evaluation, and a case study based on DiffCVSS. Note that here the information provided by DiffCVSS for the participants includes: (1) the most likely vulnerability path (the code path triggering the vulnerability); (2) the most similar vulnerability path in the Android kernel; (3) the Android functions and associated CVSS metric values.

After that, all the participants were asked to complete a questionnaire for measuring

the usability of DiffCVSS including effectiveness, precision, and satisfaction. Finally, we check their responses and perform an interview to understand their feedback of DiffCVSS in detail. We provide all the training materials and survey questions in [169].

Table 4.9: User study evaluation results. M=maintainer, S=student, A=average

	M	**S**	**A**
Manual re-evaluation time	$> 4h$ (M=8)	N/A	4.8h
Evaluation time with help of tool	21.7m	23.8m	23.1m
Reduced workload with help of tool	75.1%	78.3%	76.7%
Metric-level Accuracy	88.75%	90.31%	89.53%
Severity-level Accuracy	90%	91.2%	90.6%

Results. This study lasted over four months, including survey design, recruitment, and data collection and analysis. We elaborate on the answers to the above questions as follows:

• *DiffCVSS can save 91.98 % of time and reduce 76.7% of workload.* We demonstrate the effectiveness of DiffCVSS by comparing the re-evaluation time with and without DiffCVSS. Specifically, 90% of maintainers (M=9) were unable to manually re-evaluate the two vulnerabilities due to the time limit (120 minutes). When we asked how much time they would expect to finish the re-evaluation, 8 maintainers answered *"at least 4 hours"*, and two of them answered *"more than 6 hours"*. The average expected evaluation time is 4.8h. In comparison, with the help of DiffCVSS, most participants (M=10, S=18) successfully finished this task within 30 minutes. The average time is 23.1 min. The results show that DiffCVSS dramatically eases the vulnerability assessment for maintainers. The results are summarized in table 4.9.

Moreover, we present the responses of participants on how much and what kind of workload can be reduced with the help of DiffCVSS. The average reduction of workload is 76.7% (M=75.1%, S=78.3%). Specifically, 34.2% of participants (M=36.36%, S=32.05%) state that *"less time to find vulnerability-related call-chain"*; 28.53% of participants (M=27.27%, S=29.79%) describe that *"less time to understand the functionality of source code"*; 17.43% of participants (M=18.18%, S=16.67%) expressed that *"less time to understand the exploitability of the vulnerability"*. The results show that DiffCVSS can significantly reduce the time and efforts for vulnerability re-evaluation.

• *DiffCVSS achieves an accuracy of 89.53% in Metric-level and 90.6% in Severity-level on average.* In order to test how precisely DiffCVSS can re-evaluate vulnerabilities from user perspectives, we present each step's results of DiffCVSS to the participants. With the help of DiffCVSS, the participants are asked to re-evaluate the severity of the vulnerability in Android kernel. After that, we compare their re-evaluation results with that of DiffCVSS in Metric-level and Severity-level. Metric-level means we evaluate the fine-grained precision in terms of four exploitability metrics (AV, AC, PR, UI). Severity-level means we measure the precision in terms of a vulnerability's severity group (low, medium, high, critical) defined by CVSS [173]. Note that sometimes one metric difference will not change the severity level. The results show that DiffCVSS achieves a 89.53% of accuracy (M=88.75%, S=90.31%) in metric-level and correctly re-evaluates 90.6% (M=90%, S=91.2%) of vulnerabilities, which align with the evaluation results in §4.5.

• *The vast majority of participants expressed satisfaction with the usability of DiffCVSS.* We seek to understand how DiffCVSS satisfies the maintainers and potential users. The satisfaction metric is measured by the following key points: whether DiffCVSS can provide correct guidance and whether they are willing to utilize them for vulnerability assessment. 90% of participants (M=8, S=19) thought DiffCVSS could correctly guide them to re-evaluate vulnerabilities. One maintainer chose the option *might or might not*. He commented that *"Though DiffCVSS can help me to analyze the vulnerability, the proof-of-concept (PoC) is necessary if I want to re-evaluate a vulnerability very precisely"*. It is indisputable that PoC can contribute significant help for the security analyzer. However, only 9.7% of linux kernel related-vulnerabilities have PoC. Although DiffCVSS cannot provide the exact PoC, the 83% vulnerable call stacks it generated are the same as that of PoC (see §4.6). One participant posted a negative attitude and commented that *"Not sure whether the call-chain generated on Android kernel is correct"*. We re-checked the vulnerabilities he re-evaluated with the help of DiffCVSS and found the call-chain generated on Android indeed has a significant discrepancy with Linux, which only shares 22.53% of functions. To find the most possible vulnerable call-chain in the Android kernel, DiffCVSS matched the most similar one of the Linux kernel, which we acknowledge that it may cause a certain error. It is possible that such a vulnerability does even not exist in the Android kernel when the similarity score is very low. The security analyzer may

need extra efforts to determine in such a situation. As to the question whether they are willing to use them in the future, 90% of participants (M=8, S=19) chose *Yes*, 6.67% (M=1, S=1) chose *Not sure*, and 3.33% (M=1) chose *No*.

Analysis of False Cases of DiffCVSS in User Study. Further, we inspect why DiffCVSS failed to re-evaluate some vulnerabilities correctly. First, 38.5% of the cases are caused by uncertainty of the call-chain generation. The participants expressed doubts about the reach-ability of the call chain. One describes that *"though DiffCVSS generates the most possible exploitable call-chain, but it didn't provide enough evidence to show it is reachable which increases my uncertainty to the results of DiffCVSS"*. Second, 34.6% of cases are caused by the false positives and false negatives in function mapping. For example, the function path_get, which gets a reference to a path, is incorrectly mapped to the UI metrics. One participant said that *"when a window pops up in android apps, it may invoke the path_get, but it did not involve any user interaction"*. On the other hand, The function usb_submit_urb is a miss-reported case, which indicates it requires victims to plug a malicious USB into their computers (UI), but the UI model failed to recognize it. The others are unsatisfied with the presentation of DiffCVSS. One participant commented that *"The plain and long call-chain of Android and Linux are presented on different pages, which is hard for me to compare the difference"*. We acknowledged that the presentation of DiffCVSS results is not very user-friendly, which may impact its usability. It would be nice to visualize the call-chain and highlight the difference; however, doing so will take much laborious engineering efforts, which is not our focus in the current research stage.

4.8 Discussion

Improving the accuracy of DIFFCVSS. There are two major possible causes of false results: incorrect mapping from exploitability metrics to functions and incorrect vulnerability call-chain identification. The first issue is mainly caused by missing enough keywords and descriptions to determine the relationship between a function and vulnerability metrics. This issue can be alleviated in the future by collecting function descriptions from other sources, such as git logs that mention the functionality or design of different functions. Such information can be used to improve the precision of the model further. Also, the metric value of some functions cannot be determined only by its description, but

also its parameter. For example, the function capable(int cap) can decide if the current task has some capability in effect. Some arguments, such as cap = CAP_SYS_ADMIN, indicate an admin capability and thus indicates PR:H, but other arguments like cap = CAP_IPC_OWNER may only show a general capability and thus indicate PR:L. Therefore, future work can equip data-flow analysis or code context analysis to improve the precision of mapping functions to exploitability metrics. The second issue is often caused by the incorrect identification of indirect calls and the incomplete coverage of entry point functions written in assembly code. Correspondingly, this issue can also be alleviated by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. Equipping the end-to-end symbolic execution to verify the feasibility of call-chain can also improve the precision of vulnerability call-chain identification.

Limitation and generality. DiffCVSS still has some limitations. Most importantly, the component of mapping exploitability metrics requires well-maintained documentation that provides direct, clear, and descriptive function descriptions. Fortunately, most large open-source projects, such as the Linux kernel, which have many derivatives, are typically well maintained and thus provide enough documentation like function description. However, DiffCVSS would not work well for the projects with vague, incomplete, and inadequate documentation, which might mislead DiffCVSS and result in a wrong metrics mapping. Therefore, it will be exciting to see future work that could automatically generate the vulnerability metrics without the function description but only based on functions' semantics and their usage context. However, developing such techniques is challenging and also out of the scope of this work. Furthermore, the component of vulnerable call-chain identification is based on the program call-graph, which is built by LLVM and Clang. Thus, DiffCVSS cannot analyze the project, for which the complete call graph is unavailable, and it does not support Clang. However, in general, DiffCVSS can be applied to other open-source applications if their documentation is well maintained, and their call graph can be generated.

4.9 Related work

CVSS score/severity/exploitability prediction. Khazaei et al. [182] and Elbaz et al. [183] proposed a machine learning-based method to predict CVSS scores based

on natural language description of vulnerabilities. Han et al. [184] trained a robust deep-learning model that can extract discriminative features of vulnerability descriptions to predict multi-class severity level of software vulnerability. Many previous works [185, 186, 187] also tried to predict how soon individual vulnerabilities are likely to be exploited using features derived from vulnerability databases or social media posts. However, those approaches cannot address the "one-for-all" CVSS usage issue, because there exists no vulnerability report for different versions or derivatives (but only the mainline) to conduct those description-only analysis. Additionally, there are some program analysis-based methods that infer the exploitability of vulnerability. However, those works are less scalable and applicable due to the requirement of PoCs/exploits or using less-generic self-defined metrics.

Vulnerability severity rating. Numerous works also try to rate the vulnerabilities to help patch prioritization and evaluate the severity of vulnerabilities. CVSS [121] generally discussed the Common Vulnerability Scoring System. Liu et al. [188] compared existing vulnerability severity scoring systems X-Force, CVSS and VRSS, based on which they also provide their own vulnerability scoring system. Han et al. [184] provide a system based on word embeddings and a CNN, which can capture special word and sentence features from vulnerability descriptions and further use them to predict vulnerability severity. Similarly, Spanos et al. [189] also provide a vulnerability severity scoring system based on text mining against the description of vulnerabilities. However, most of these existing works are only based on textual information of vulnerabilities, which are typically limited to the specific version and vendor of a project. Thus, these works cannot address cross-environment vulnerabilities effectively.

Patch prioritization. A widely regarded principle is that security-critical bugs should be prioritized for patching. Many previous works [84, 80, 83] thus try to identify security-critical bugs from general bugs through machine learning techniques. Arora et al. [190] provide an empirical study on vulnerabilities disclosure, which shows that vulnerability disclosure can accelerate patch release, and vendors are more responsive to more severe vulnerabilities. VULCON [191] is a vulnerability management strategy, which can prioritize vulnerabilities for patching based on the input that includes a series of vulnerability reports, asset criticality, and personnel resources. Sharma et al. [192] leverage word embedding and convolution neural network (CNN) to prioritize

vulnerability by analyzing the vulnerability description. However, these works are only based on the description information from the CVE or patches, which may not be precise enough when the description only includes limited information. Furthermore, none of these existing works can analyze the severity of cross-OS vulnerabilities. Unlike these works, DiffCVSS not only analyzes the description information but also the exploitation information collected from call chains using program analysis techniques, and thus is more precise and can address the cross-OS situation.

Vulnerability in cloned projects. Due to the code clone/reuse, many vulnerabilities propagate to multiple projects. Some previous works try to identify these vulnerabilities. VulSeeker [193] and Gemini [194] are based on machine-learning techniques, which can analyze the similarity of code and check the existence of cross-platform vulnerabilities in binary code. XMATCH [195] detects cross-platform vulnerabilities in embedded systems and IoT devices based on extracting and comparing conditional formulas as semantic features from the binary code. Some previous works also conduct empirical studies about the severity and influence of vulnerabilities that propagate to different projects. ADDICTED [196] reveals the security risk brought by software shipment and customization, as vendors and carriers enrich the system's functionalities without fully understanding the security implications. Farhang et al. [197] empirically studied the security bulletin from Android and three leading vendors: Samsung, LG, and Huawei. Their results show that vendors would evaluate vulnerabilities and react with CVEs with Android Git repository references without delay. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [135] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DiffCVSS, after pointing out or discovered the issues caused by vulnerabilities that influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [135] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DiffCVSS, after pointing out or discovered the issues caused by vulnerabilities that

influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities.

Chapter 5

GNNIC: Finding Long-Lost Sibling Functions with Abstract Similarity

Large programs, such as operating system (OS) kernels, often use indirect calls to achieve dynamic and polymorphic behaviors. For example, the Linux kernel 5.7 and Android kernel 5.10 have more than 55K and 62K indirect call sites. Accurately identifying these indirect calls is critical for building precise call graphs and control flow graphs, which are the basis of modern program analysis and security analysis, including bug detection [198], program debloating [199, 200], directed fuzzing [201, 202], vulnerability assessment [203], model checking [204], and many others [205, 206, 88, 207].

To identify indirect-call targets, recent works are primarily based on the following three methods: (1) Refining indirect call targets based on type analysis [208, 108]. Type analysis is widely used in indirect-call target identification because it is simple and can be sound in most cases. Specifically, it refines the indirect call targets by matching the type information of potential target functions and the function pointers. (2) Identifying and refining indirect call targets based on point-to analysis [209], which is a general approach to matching the pointer to its pointed addresses. (3) Refining the indirect call targets at runtime using control-flow integrity (CFI) [210] or other dynamic techniques [211].

However, the results of traditional approaches are still imprecise. According to our evaluation (see §5.5.4), the precision rates for type-based approaches are typically less than 10%. This is mainly because type-based methods rely solely on type-related constraints

for matching, without considering code semantics or behaviors. This issue is even more pronounced in large programs or when the type is too general. For instance, consider the function pointer $int(*console_blank_hook)(int)$ in the Linux kernel V5.7, which has an *int* argument and an *int* return value. The type-based approaches match more than 1,200 target functions. However, in fact, there is only one real target function, $apm_console_blank$ in the kernel. All other matches are false positives that merely share the same function type as this particular target. On the other hand, point-to-based approaches can suffer from both precision and recall issues. This is because precise pointer analysis is still an open problem for large programs. At last, indeed, dynamic analysis is precise, but its recall issue would make it less useful for analyzing OS-level large programs. According to the results of HFL [212] and DR.FUZZ [213], which are two recent kernel fuzzers, the code coverage is typically less than 10%, and even less than 1% for drivers.

Observations. Through an empirical study, we found that target functions of an indirect call typically share the same abstract behaviors (the most relevant behaviors to the major functionalities of a function), despite having different detailed behaviors due to polymorphism. We name the similarity across target functions as *abstract similarity*. For instance, in an operating system, there may be tens of file systems, each with its own implementation of the *write* function. The *write* functions are typically called through an indirect call based on the actual file system used. Although these *write* functions differ, they share the same abstract behavior — writing an amount of data in a buffer to a file specified by a file descriptor. Therefore, when a known target exists for a specific indirect call, which is a practical assumption, as demonstrated later, abstract similarity can be used to identify other targets in the program.

One question that arises immediately is what kind of information can describe the abstract behaviors of a target function. Our empirical study shows that one direct source of information we can use is the representative information and the metadata of a function, which includes the function description, signature, and data types. In addition, we found that nested function calls also contribute to determining the abstract behaviors. We have further measured the significance of each kind of information in determining the abstract behaviors. In this project, we refer to such information that can represent the abstract behavior of functions as *abstractive information*.

Additionally, we found that graph structures are well-suited to represent abstractive

information. This is because code semantics have been commonly represented in the form of graphs, such as call graphs and control-flow graphs. Furthermore, additional textual information, such as function names and descriptions, can be integrated into the graph nodes to capture and represent the human-understandable functionalities of functions. In the case of abstractive information, a graph structure can also accurately capture function invocations, data types, their relations, and textual descriptions.

In this paper, we propose using graphs to represent abstractive information and utilize Graph Neural Networks (GNNs) automatically learn and aggregate the abstractive information for deriving the abstract behaviors of functions. Our approach, named GNNIC, stands for Graph Neural Network (GNN) based Indirect Call analysis. By conducting the abstract-similarity analysis between the known target function of an indirect call and other functions, we can utilize GNNIC to identify more target functions for the indirect call.

Technical challenges. We anticipate two main challenges in implementing the GNNIC approach. First, we require at least one confirmed target function (referred to as an "anchor function") to start the matching process for more targets. Collecting anchor functions is challenging because static techniques often produce false positives, while dynamic-based methods have low coverage rates. Second, we need a reliable and versatile technique to represent abstractive information and build a precise similarity analysis for it. The abstractive information is diverse, including textual descriptions, data types, and function calls. We need a new representation technique to generally process such diverse information.

Key techniques. To address these challenges, we propose the following techniques. First, to identify anchor functions, we develop *scoped unique-name matching*. The technique is based on the fact that if a function pointer's name is unique in its dependency scope (i.e., possible scope in which the function pointer can be defined), any function assigned to the function pointer is most likely a valid target of an indirect call using the function pointer based on the same unique name. The technique thus focuses on generally identifying unique names and delimiting dependency scope. Second, we propose the use of a representative abstraction graph (RAG) that captures the diverse abstractive information of a function, including the descriptions and signatures of functions, the nested function calls, and data types. GNNIC trains a graph neural network against the

RAG and generates embeddings for every function, based on which GNNIC can compare the abstract similarity between functions and anchor functions to match more targets.

We have built GNNIC upon LLVM, StellarGraph [214] and GraphSage [215]. We evaluated the effectiveness and performance of GNNIC with the commonly used large programs. We select OS-level programs, as they represent the largest and most complex programs. Experimented programs include the Linux kernel, Android kernel, and FreeBSD kernel. The evaluation results show that compared with state-of-the-art techniques, GNNIC can further improve the overall precision from 10% to 92.3%. Such an improvement is significant; for example, to identify a triggerable call chain in the Linux kernel through static analysis, on average, we can reduce the number of detected call chains from 10^9 to 10^4 based on the results of GNNIC (see §5.6). Furthermore, GNNIC enhances traditional program analysis by reducing false positives and detecting previously missed new bugs.

5.1 Background and Study

Large programs usually use indirect calls to increase the flexibility and scalability of C and C++ code. In this study, we employ the illustrative example presented in Figure 5.1 to demonstrate the indirect calls. To complete an indirect call, the process includes (1) taking the address of a function (target); (2) storing the address to a function pointer; (3) propagating the function pointer to an indirect call site; (4) dereferencing the function pointer and calling the target.

5.1.1 State-of-the-Art for Detecting Indirect-Call Targets

In theory, both point-to analysis and type analysis can be used to analyze indirect calls. However, due to the precision and scalability issues of the global point-to analysis, point-to-based approaches are not commonly used for identifying the indirect call targets in large programs. Therefore, the state-of-the-art works [216, 217, 208, 218, 108, 219] use type-based approaches to handle indirect calls in programs. In this section, we discuss their limitations.

False positives resulted from generic types. Due to the following two reasons, applying the type-based analysis to large programs may trigger a large number of false

positives. The first cause is related to generic types. When the types of function pointers are too general, such as $int(*console_blank_hook)(int)$, type analysis often performs poorly and falsely matches hundreds to thousands of targets. MLTA [108, 208] tries to alleviate this problem by involving more layers of type constraints, but it still struggles with generic types, often reporting thousands of targets for a single indirect call.

False positives resulted from global search. Another common problem with traditional type-based approaches is that they globally search for matched targets in the whole program. As type analysis does not track data flows, such a global search becomes a must, which, however, incurs many false positives. In a large program, many modules are not dependent on each other. For example, the Linux kernel has 15 well-defined subsystems, each containing numerous independent modules. Functions in modules that are independent of the module containing an indirect call can never become valid targets of the indirect call, even if their types match. Unfortunately, type matching is unable to delimit the search scope, which results in many false positives.

Out-of-the-scope cases of MLTA. To understand the cases that MLTA cannot handle, we have reused the two-layer type analysis implementation from [208] against the Linux kernel v5.7. The results show that many indirect calls cannot be handled by MLTA and must resort to single-layer type matching. There are two out-of-scope cases for MLTA. The most obvious case is that many function pointers do not involve a composite type (e.g., struct), i.e., they have a single layer of type, thus, cannot benefit from MLTA at all. Second, function pointers or object pointers are frequently cast to general-type pointers (e.g., $void*$), which also makes the function pointers disqualified for MLTA. Therefore, MLTA generates, on average, 84.7 potential targets for indirect calls in the Linux kernel, which is still very large, considering there are more than 55K indirect call sites in the Linux kernel.

5.1.2 An Empirical Study of Abstract Behaviors

The abstract behaviors of target functions. We refer to a *behavior* of a function as an operation against an object. Then, the abstract behaviors of a target function are supposed to be the most relevant behaviors to the major functionalities of the function. In the example of indirect call $hw- > mac.ops.check\_for\_link(hw)$ (see line 38 in

```
/*****************Potential Target functions*****************/
s32 e1000_check_for_copper_link_ich8lan(struct e1000_hw *hw) {
    struct e1000_mac_info *mac = &hw->mac;
    s32 ret_val, tipg_reg = 0;
    ...
    ret_val = e1000e_phy_has_link_generic(hw, 1, 0, &link);
    ...
    e1000e_check_downshift(hw);
    ...
    ret_val = e1000e_config_fc_after_link_up(hw);
    if (ret_val)
        e_dbg("Error configuring flow control\n");
    ...
}
s32 e1000e_check_for_copper_link(struct e1000_hw *hw) {
    struct e1000_mac_info *mac = &hw->mac;
    s32 ret_val;
    ret_val = e1000e_phy_has_link_generic(hw, 1, 0, &link);
    ...
    e1000e_check_downshift(hw);
    ...
    ret_val = e1000e_config_fc_after_link_up(hw);
    if (ret_val)
        e_dbg("Error configuring flow control\n");
    ...
}
/******************Function address taken******************/
static const struct e1000_mac_operations ich8_mac_ops = {
    .check_for_link     = e1000_check_for_copper_link_ich8lan,
    ...
}
s32 e1000_init_mac_params_80003es2lan(struct e1000_hw *hw) {
    mac->ops.check_for_link = e1000e_check_for_copper_link;
}
/********************Indirect call site********************/
static bool e1000e_has_link(...) {
    ...
    ret_val = hw->mac.ops.check_for_link(hw);
    ...
}
```

Figure 5.1: An indirect-call example in the Linux kernel.

Figure 5.1), we can list and empirically rank the behaviors of each target function based on their relevance to its major functionalities. The most relevant behaviors of $e1000\_check\_for\_copper\_link\_ich8lan()$ include: checking the existence of the link, checking if there is $downshift$, configuring the link and checking configuration status, and checking and handling the link for different mac types; while the most relevant

behaviors of $e1000e\_check\_for\_copper\_link$ include: checking the existence of the link, checking if there is $downshift$, configuring the link, and checking the configuration status. By cross-checking the targets, we can clearly see that they indeed share three most relevant behaviors. In other words, these two targets share their abstract behaviors.

Furthermore, we have conducted a manual analysis of 501 target functions from 100 random indirect calls to verify if abstract behaviors are commonly shared between target functions of indirect calls. Our analysis showed that all indirect calls have targets that share at least one relevant behavior, with commonness decreasing as relevance decreases. This confirms that the most relevant behaviors, i.e., the abstract behaviors, are commonly shared across indirect call targets.

The abstractive information of target functions. Next, we aim to understand what information can be used to represent the abstract behaviors of a function. In pursuit of this goal, we have examined 100 indirect calls and their corresponding target functions, as mentioned in the previous subsection for our empirical investigation. It is important to note that the findings presented in this section serve only as empirical or directional guidance. Additionally, our sampling methodology and its statistical validity are thoroughly addressed in Section §5.5.1. Specifically, we first select possible behavior-related information, including the name of the target function, the data types used by the target function, the nested function calls, the control flow of the target function, and the data flow of the target function. We then empirically analyze which information is commonly shared across targets. The idea is that, as abstract behaviors are shared, we believe that commonly shared behavior-related information among target functions can be likely used to describe abstract behaviors.

The results show that the target functions for 97% of the indirect calls share similar names and descriptions and correspond to the behavior indicated by the function pointer name. For instance, as illustrated in Figure 5.1, the target functions of the indirect call exhibit a comparable function name, "check for link," signifying the analogous behavior of these functions. Additionally, we discover that the nested function calls and data types of target functions within an indirect call display similarity, boasting average Jaccard similarities of 0.62 and 0.41, respectively. This is determined by comparing the callee sets and utilized type sets of target functions. These values are markedly greater than the average similarity between randomly chosen functions, which are 0.33

and 0.23, respectively. Moreover, upon conducting a manual comparison of control and data flow among the target functions of a specific indirect call, we observe that only around 38% and 16% of them display similarity. To conclude, we can use function names and descriptions, internal function calls, and data types as the abstractive information of target functions.

Using GNN to represent nested abstractive information. Manually analyzing abstract behaviors is relatively straightforward for experienced programmers since people can understand and learn the descriptions, names, and nested callees of functions. However, automatically summarizing such abstract behaviors is not easy. This is because function calls and types are nested in programs. Suppose the tool cannot analyze such a nested relationship, in that case, it cannot accurately capture the abstractive similarity of functions. For example, *writeb* and *writew* further call $\_\_raw_w riteb$ and $\_\_raw_w ritew$ respectively. The tool can accurately capture the similarity between *writeb* and *writew* only if it can understand the names of these functions and can further collect the abstractive information from their callees. Then, the tool can reflect all this abstractive information to the abstractive similarity between *writeb* and *writew*.

GNN is well-suited to catch such abstractive information of the nodes in the graph. The intuition of GNN is that given a node in the graph, GNN can aggregate the information of adjacent nodes to represent the given node. The neighboring nodes can further be described by their neighboring node. This feature is excellent for describing the nested abstractive information of functions in the call graph, because given a function in the call graph, we want to aggregate the abstractive information from its callee, which can further be used to describe the abstractive behavior of the given function. Similarly, these callees can be further described by the abstractive information of their callees. Therefore, GNN can accurately catch the abstractive behaviors of functions as we expect.

5.2 Overview

5.2.1 The Workflow of GNNIC

The goal of GNNIC is to identify indirect-call targets precisely, even for large programs. Figure 5.2 shows the overview workflow of GNNIC, which consists of four parts: ①

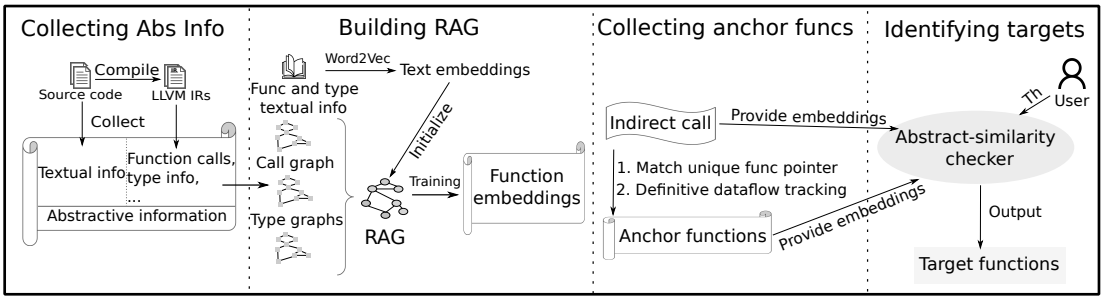

Figure 5.2: Overview of GNNIC. RAG: representative abstraction graph, Abs Info: abstractive information, Th: threshold specified by user.

collecting abstractive information, ② building representative abstraction graph (RAG) as GNN inputs, ③ collecting anchor functions for each indirect call, and ④ identifying more target functions using abstract similarity.

Specifically, by analyzing the source code and LLVM IRs of the target program, GNNIC first collects some basic information, including the representative information (e.g., function names, function descriptions, etc.), indirect call sites, all potential target functions, type-related information, the program call graph (without considering indirect call), etc. Then, GNNIC builds a RAG that integrates the call graph, representative information, and type-related information. It uses the RAG to train a graph neural network (GNN) model, based on which GNNIC can get a unique embedding for every function in the program. Furthermore, with a new technique, scoped unique-name matching, GNNIC identifies anchor functions for each indirect call in a delimited searching scope. At last, given an indirect call and its anchor functions, GNNIC uses the abstract similarity to identify more target functions.

5.2.2 Challenges and Techniques of GNNIC

Before showing the design of GNNIC, we first discuss the technical challenges (C) of realizing the GNNIC approach and the corresponding techniques (T) to address these challenges.

C-1: Collecting the anchor functions. To use abstract similarity to match more targets, GNNIC first requires anchor functions—at least one validated target function for an indirect call, which is still non-trivial. When talking about collecting real target functions, the most intuitive idea is using dynamic analysis because it would typically

not introduce false targets. However, dynamic analysis against the system-level large programs suffers from a very low coverage rate. Thus, it is not a good solution for GNNIC. Additionally, preparing a fuzzing environment for a new program requires much effort and is time-consuming, such as generating quality seeds and preparing specific hardware for the driver. On the other hand, as we discussed, none of the current data-flow-based or type-based approaches can guarantee the precision of their indirect-call-targets identification. Therefore, new techniques are required for collecting anchor functions.

C-2: Digesting diverse abstractive information for similarity analysis. Given an indirect call, even if we have its anchor functions, it is still challenging to compute abstract similarity due to the diverse nature of abstractive information of target functions. Such information includes textual information, code semantics, and types. In addition, the relations among different kinds of information are also complex. Functions can call each other; complex types can contain each other; and functions can use different complex types. Thus, we need a systematic way to integrate diverse information, which should also be compatible with GNN.

T-1: Scoped unique-name matching for anchor functions. To address C-1, we propose a new technique to precisely identify anchor functions. In this step, we aim for high precision but allow for false negatives in identifying targets. Our technique is based on the fact that if a function pointer's name is unique in its dependency scope (possible scope in which the function pointer can be defined), any function assigned to the function pointer will likely be a valid target of an indirect call dereferencing the function pointer (with the same unique name). To realize the technique, we generalize the definition of "name" to support function pointers in structs and employ an iterative algorithm to narrow down the dependency scope for an indirect call. Our evaluation shows that the technique can identify at least one anchor function for 93.7% of indirect calls with a precision of 94%. We present the details of the technique in §5.3.3. In addition, we find two scenarios where we can definitively determine anchor functions: callback functions and global variable related indirect calls. Thus, we also develop a definitive data flow tracking to precisely identify anchor functions for such indirect calls.

T-2: Catching diverse information with representative abstraction graph. To digest the diverse information about the abstract behaviors of a function, we propose the *representative abstraction graph* (RAG). RAG is not only capable of integrating

various kinds of information, such as textual description, but, more importantly, it is also compatible with GNN. Its graph structure can be seamlessly processed by GNN. Specifically, RAG is essentially a graph structure in which each node is a function or type augmented with its representative information, and each edge represents the relation between nodes. By representing the abstract behaviors of functions with RAG, we are able to train a graph neural network for generating embeddings for abstract behaviors, which finally allows us to match more target functions from anchor functions based on the abstract similarity.

5.3 The design of GNNIC

5.3.1 Collecting Abstractive Information

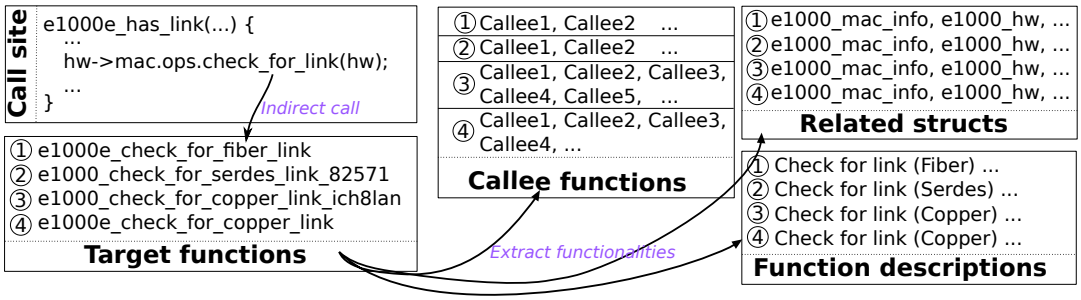

Figure 5.3: Abstractive information of target functions.

As shown in our empirical study in §5.1.2, there are mainly three types of abstractive information that can reflect abstract behaviors of target functions: (1) function names and textual descriptions, (2) the function calls, as well as their relations, and (3) data types that are used by the functions. More specifically, the function names and descriptions are expected to be descriptive, which are intended to help programmers quickly understand the major functionalities of functions. Undoubtedly, they are important information that should be included for describing abstract behaviors. To a great extent, the nested calls of a function can also determine its abstract behaviors. This is because code reuse is a programming paradigm. In almost all programs, primitive functions (e.g., library functions for allocation, file operation, and memory management) are frequently reused. In other words, the parent function often acts like a synthesizer that logically organizes

the primitive functions. Therefore, nested calls and their relations should also be included to describe the abstract behaviors of a function. Furthermore, the data types are the metadata pre-defining the objects that are operated by function behaviors, so they should also be included.

As an example, Figure 5.3 summarizes the abstractive information of several target functions for the indirect call, $hw->mac.ops.check\_for\_link(hw)$ (also see line 38 in Figure 5.1). First, the function names and descriptions of these target functions are descriptive, indicating the abstract behavior that checks the status of some links in the $e1000$ module. Moreover, the shared callees of these target functions, such as $e1000e\_config\_fc\_after\_link\_up$ and $e1000e\_phy\_has\_link\_generic$, and the data types used by these target functions including $e1000\_mac\_info$ and $e1000\_hw$ also indicate that the abstract behaviors of these target functions are $e1000$ and "link" related. Therefore, all such information is included as abstractive information.

Collecting abstractive information of target functions. We can collect the data types and function call information through an analysis pass on LLVM IR and collect the textual information from the source code. Specifically, by analyzing IR, GNNIC first extracts the call relations between functions, the types used by functions, and the inclusion relations between types. Furthermore, GNNIC uses regex expressions to collect the descriptions of functions and types by scanning the source code.

5.3.2 Building RAG as GNN Inputs

As the next phase of GNNIC, it generates a vector for every function of the program, which can be used to evaluate the similarity of functions. Specifically, it includes three parts: (1) using vectors to represent the textual information, (2) generating representative abstraction graph (RAG) and, (3) leveraging the RAG to generate function embeddings.

Embedding textual information for the representation of functions and types. The names and descriptions associated with functions and types are considered as textual descriptions. Such descriptions are usually human-readable and encompass key information for understanding the functionalities. Thus, such information can assist GNNIC in evaluating the abstract similarity of functions. However, in order for programs to understand such textual information, we need natural language processing (NLP)

techniques to turn such textual information into vectors. For example, by examining the description, the programmer can see that the "release" operation is similar to the "free" operation, which, however, is not quantifiable for programs. But after applying the NLP techniques, such two words can be changed into vectors with a short distance and thus can be quantified by other programs. Therefore, in this project, to easily integrate textual information into RAG and then be analyzed by GNN, GNNIC first equips NLP techniques to embed such textual information into vectors. Specifically, for each function and type, GNNIC separates its name into tokens and combines them with the descriptions (if any) as a single sentence. For example, for the function e1000e_check_for_fiber_link, GNNIC extracts "e1000e", "check", "for", "fiber", and "link" from its function name and also collects the comment *Check for link (Fiber); Checks for link up on the hardware. If link is not up and we have a signal, then we need to force link up.* from its function description. Notice that we only consider the overarching descriptions of functions, excluding comments embedded within the functions' code. These internal annotations, while often elucidating specific variables or detailed operations, do not typically serve as summary descriptions of the whole function. GNNIC then pre-processes such text information based on the commonly used text pre-processing techniques [220], such as removing stop words and stemming. At last, by using Word2Vec [221, 222], the corpus of each function and type is embedded into a vector with 300 dimensions, which are used as the initial features for functions and types in RAG. To do so, GNNIC pre-trains a Word2Vec model based on more than 1.5GB textural corpus, which combines the code comments, documentation, git logs, and other program-related texts from multiple git repositories, including Linux, FreeBSD, OpenSSL, etc. In this way, the model can learn more descriptions and words related to programs.

Representative Abstraction Graph (RAG). Since there are three types of representative information, treating them separately can only yield one-sided results. Such information thus should be integrated at first. To this end, we propose RAG to leverage the graph structure reflecting the use, invocation, and containment relations between types and functions. Simultaneously, it incorporates individual node information, such as names and descriptions. To integrate such information, GNNIC first builds a separate graph for each relation between nodes (including nested functions and types) and then merges them into one graph, which is RAG. Finally, GNNIC initializes each node with

its corresponding textual information.

Specifically, GNNIC first builds a directed graph to represent the function call graph (without considering indirect calls), indicating the function call relationship. Then, GNNIC builds a type usage graph that records all the functions and data types used by these functions. In this graph, the nodes are functions and types, and the directed edges point from functions to types indicating the usage relation. At last, GNNIC builds the type-relation graph, which describes the containment relations between different types. Each node is a specific data type, and the weighted directed edges point from the container types to its element types; the weights are the counts for the number of the specific element's types. For example, as a struct type, $structe1000\_mac\_info$ can be one of the nodes in the type-relationship graph, which includes one element with type $structe1000\_mac\_operations$, three elements with type $u8$, twelve elements with type $bool$, etc. Therefore, in the type-relationship graph, from the node $structe1000\_mac\_info$, there are at least three edges, an edge points to $structe1000\_mac\_operations$ with weight 1; an edge points to $u8$ with weight 3; and an edge points to $bool$ with weight 12. Based on all these relationship graphs and the embedding of textual information, GNNIC then merges them into a RAG.

As shown in Figure 5.4, this merged graph is RAG, in which the nodes are functions and types with their embedding. Edges in the RAG can point from function to function, function to type, or type to type, which represents the relations between the caller with the callee, the type-user with the type, and the type with the types of its elements. After this, GNNIC trains a GNN against this RAG.

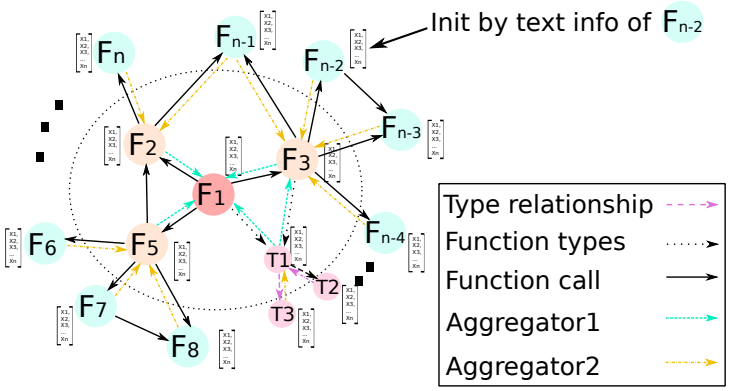

Figure 5.4: Structure of GNN on representative abstraction graph. F=function, T=type.

Generating node embedding for RAG. This step aims to use an unsupervised approach to learn embeddings of functions and types only based on the RAG. To this end, we choose to use the state-of-the-art technique, GraphSage [215], to embed nodes in the RAG. We choose GNN mainly because it can effectively learn and utilize the relations between nodes in the RAG. In particular, when given a node in the graph, GNNIC aggregates the feature information of its neighbors and represents this node with the aggregated information. Given a node v in k-th layer, its embedding h_v^k can be expressed by the nodes in the (k-1)-th layer as follows:

$$h_v^k = \sigma(\mathbf{W} \cdot Mean(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in N(v)\})) \tag{5.1}$$

Here, $N(v)$ represents the sampled neighbors of node v; $\mathbf{W}$ indicates the trainable weight matrix; σ is a non-linear activation function such as ReLu. For example, as shown in Figure 5.4, the embedding of the function $F1$ is generated by the aggregation of the abstractive features for its callee functions $F2$, $F3$, $F5$, and the contained type $T1$. Based on this approach, GNNIC can generate the embedding for every function in the program, representing the abstract behavior of functions.

5.3.3 Collecting Anchor Functions with Scoped Unique-Name Matching

As we discussed in §5.2, GNNIC should first collect anchor functions, with which GNNIC can further refine the indirect call targets by checking the abstract similarity between potential target functions and these anchor functions. The goal is to precisely identify at least one anchor function for as many indirect calls as possible. It is worth noting that this step never aims for completeness but for precision. Also, as we discussed in **C1** (see §5.2.2), due to the low coverage rate or high false-positive rate of traditional approaches, we need a new solution.

Intuitively, if a function pointer's name is unique in the program or in the scope of its dependencies, we can precisely match any assignment of a function's address to the function pointer with an indirect call using the function pointer based on the name, without the need for tracking the data flows. Any function whose address is assigned to a unique function pointer is most likely a valid target of an indirect call. Based on the

insight, we propose *scoped unique-name matching* to identify anchor functions. Although in theory, it is possible that the function pointer can be re-defined, resulting in invalid matched targets, our evaluation in §5.5.3 shows that such a technique works extremely well for GNNIC—it can identify at least one anchor function for 93.7% of indirect calls with 6% false positives.

For example, in the Linux kernel, the name of function pointer *associate_indicator* is unique in the whole kernel. Also, we can find that the address of the function *lane2_assoc_ind* is assigned to this unique function pointer. Therefore, the function *lane2_assoc_ind* is an anchor function for any indirect call that uses this function pointer. Note that because of the uniqueness of the function pointer, we can know if an indirect call dereferences the function pointer. The power of such a unique name matching is that it eliminates the need for data flow tracking from function assignment to indirect call.

To make the technique general and precise, we still need to overcome two problems: (1) generalizing the definition of "name" for function pointers and (2) delimiting the dependency scope for an indirect call.

Defining unique names. In large programs, function pointers typically exist as fields of structs instead of as standalone pointers. Therefore, we define the name as a *composite name* that includes the name and type of struct objects, as well as the name of the function pointer. For example, for the function pointer on line 38 of Figure 5.1, its composite name is a combination of strings, "e1000_mac_operations", "ops", "s32*(struct e1000_hw *)", and "check_for_link", which contain the name and type of the function pointer, as well as its struct object. Such a representation of the name allows us to generally cover all kinds of function pointers.

Delimiting the dependency scope. Another problem we overcome is the scoping of dependencies. Apparently, indirect calls can only call the functions of modules that have data dependencies with the indirect call, as the addresses of the functions must be passed to the indirect call through data flows. Therefore, in this step, we delimit the scope to only data-dependent modules for the indirect call. We choose to perform the scoping at the granularity of the module to simplify the design and ease the implementation, which has worked reasonably well.

Specifically, GNNIC adopts a simple yet effective policy to define dependency. Two

modules can have data dependencies only under the following two conditions, ① one module calls functions defined in another module, or ② one module uses the global variable defined in another module. Based on these two conditions, by iteratively considering modules with data dependency as the scope, GNNIC uses unique-name matching to identify anchor functions only within that scope. Notice that, to be precise, when finding the data dependency modules, GNNIC does not consider the data flow passed by indirect calls. Such a limited scope can significantly narrow down the search space, so that the name of a function pointer has a high chance to be unique. For example, $ops.release()$ is an indirect call in $e1000e$ module. In the whole kernel, we can find hundreds of target functions whose addresses are assigned to the function pointers with the same name. However, after we delimit the scope to the modules that have data dependency with $e1000e$, only eight targets are identified and all of them are valid according to our manual analysis.

```
Type ST GV{
    .func_pointer = foo1;
}

void CallerOfIcalls(Type (*FuncPointer)(Type A), ...) {
    FuncPointer(A);
    GV.func_pointer(...);
}

void CallerOfIcaller(...) {
    CallerOfIcalls(foo2,...);
}
```

Figure 5.5: A simplified example for definitive data flow tracking.

Definitive data flow tracking for function pointers. In addition to using scope unique-name matching, we find two scenarios in which standard data flow tracking suffices to identify anchor functions: (1) call back as a function argument, and (2) direct use of a function pointer in a global initializer.

Specifically, for the call-back functions, the function address is typically directly stored to a function pointer as an argument, which is typically used in the current function. For example, in Figure 5.5, on line 11, the function address of $foo2$ is directly passed to $CallerOfIcalls$ and used in it. Through an intra-procedural backward data flow analysis, GNNIC can precisely know that $foo2$ is an anchor function.

On the other hand, a global object is often initialized through a *global initializer* that clearly stores the address of a function to a function pointer in the object. When an indirect call uses the global, we can precisely know the function is an anchor function. For example, in Figure 5.5, $func\_pointer$ in the global variable GV is initialized as $foo1$ (on line 2). Thus, $foo1$ is an anchor function for the indirect call on line 7. Our evaluation shows that GNNIC can use such definitive data flow tracking to find anchors for 2% of indirect calls.

5.3.4 Identifying More Targets with Abstract Similarity

After we get the embedding of each function and the anchor functions of each indirect call, GNNIC then tries to identify more target functions using abstract similarity.

Similarity analysis. Since we have the embedding of abstract behaviors for both anchor functions and other functions, the idea is to compute the similarity of embeddings. Given an anchor function (or sometimes multiple anchor functions) and a to-be-matched candidate function, GNNIC evaluates the similarity based on the following expression.

$$S_c = MEAN(\{\frac{\mathbf{v}_a \cdot \mathbf{v}_c}{\|\mathbf{v}_a\|\|\mathbf{v}_c\|}, \forall \mathbf{v}_a \in G(\mathbf{v})\}) \tag{5.2}$$

Here, let us assume $func\_t$ is a candidate target function. Then, $\mathbf{v}_c$ is the embedding for the $func\_t$; S_c represents the similarity between this function and the anchor functions; $\mathbf{v}_a$ is the embedding for an anchor function; $G(\mathbf{v})$ represents the set for all anchor functions of the indirect call. GNNIC uses the average number for the similarity between $func\_t$ and all anchor functions of the indirect call, which can show the abstract similarity of $func\_t$ with the ones of anchor functions. The similarity, S_c, always belongs to the interval $[-1, 1]$; the larger S_c is, the more similar $func\_t$ and the ground truth are. Therefore, $S_c = 1$ means $func\_t$ has the same functionality with at least one anchor function. Thus $func\_t$ very likely belongs to the real target function of the indirect call. Oppositely, $S_c = -1$ means $func\_t$ is totally different from all the anchor functions. Thus $func\_t$ unlikely belongs to the real target function of the indirect call.

This approach makes the inputs and outputs of GNNIC flexible. First, GNNIC can refine the results of different traditional indirect-call analysis techniques such as type analysis and multi-layer type analysis. Second, GNNIC can easily adjust the

false-positive rate and false-negative rate of the results by changing the threshold (S_c) to fit the precision-driven applications or the recall-driven applications. Third, our evaluation results reveal a notable similarity among anchor functions involved in indirect calls with multiple anchors, as evidenced by a medium similarity of 0.79 and an average of 0.78. Such a feature enables incorporating multiple anchor functions to ensure system robustness with minimal false positives, providing reliable and accurate results. Note that GNNIC is complementary to existing approaches. For example, it can be used to refine the results of type analysis by further removing functions reported by type analysis that are not similar to anchor functions in abstract behaviors.

5.4 Implementation of GNNIC

We have implemented GNNIC based on LLVM and GraphSage [215]. This section presents several implementation details of GNNIC.

Extracting the name of function pointers. As discussed in §5.3.3, in order to collect anchor target functions, GNNIC collects and matches the unique function pointers in a limited scope. Collecting the name of the function pointer and container struct at the indirect call site is done by a Python script code, including mainly two parts, collecting function-pointer initializers and expanded Macros. Here we did not directly work on the pre-compiled code, in which the Macros are expanded. This is mainly because we can only map the instructions in the LLVM IR into the source code lines based on the debug information. However, we cannot directly map such instructions to the pre-compiled code. Specifically, given an indirect call instruction, GNNIC first collects its source code line based on the debug information provided by LLVM. Further, GNNIC checks if a Macro does have such an indirect call. If the answer is yes, GNNIC further looks for the function-pointer name and struct name inside the Macro. Otherwise, GNNIC directly extracts such names. This approach works well for most indirect calls; however, it cannot correctly handle some corner cases, such as the function pointer name being formed by multiple strings pasted by token-pasting operators.

Handling no anchor function cases. Based on our evaluation (see §5.5.3), GNNIC can detect at least one anchor function for 93.7% of indirect calls. For the uncovered 6.3% of indirect calls, we choose to "borrow" the anchor functions from similar indirect

calls to represent their abstract behaviors. Specifically, given an uncovered indirect call, GNNIC first finds another indirect call that is most similar to it by comparing the name and type of the function pointer. Then, GNNIC regards the anchor functions of that similar indirect call as the anchor functions for the uncovered indirect calls. This decision is based on the results of the background study, which show that in most cases, the name of function pointers is highly related to the abstract behavior of the indirect call. Therefore, similar indirect calls also tend to have similar function pointers and abstract behaviors. This is essentially another application of abstract similarity, where it is applied to indirect calls instead of functions. More evaluation related to the anchor functions can be found in §5.5.3.

Handling RAG with indirect callees. When building the RAG, GNNIC incorporates the call graph, which is part of the abstractive information we include. A target function may have many nested calls, and some of them are indirect calls. When a nested call is an indirect call, the nested callee is missing and cannot be incorporated into the abstractive information of the target function. To handle this problem, we borrow the identified anchor functions of indirect calls to complete the missing nested callees. Such a strategy can slightly improve the overall stability of the model when we use RAG to train the function embeddings.

Setting up the GNN model. We use the directed GraphSage [215] model, which is implemented by the library StellarGraph [214], to handle RAG. Specifically, we use two layers in the GraphSage encoder, with a 10 and 5 sample size for the first and the second hops. We run five epochs with batch size 10K. The input of the mode is RAG, the nodes of which are initialized by the vector representing the textual information with 300 dimensions.

5.5 Evaluation

5.5.1 Experiment Setting and Data Sets

Platform. We use two computing resources available to us, including a server (8 cores/60GB memory/a single GPU) and a desktop (24 cores/64GB memory/a single GPU). Both of these two machines are running on Ubuntu 20.04.

Ground-truth targets for false-negative evaluation. To evaluate and compare the false-negative of GNNIC under different settings, we must first have a set of ground-truth targets. To this end, we downloaded and analyzed the previous fuzzing logs from the results of Syzbot [223], from which, we identified 3,831 unique indirect caller-callee pairs. To acquire these function pairs, we first extracted all caller-callee pairs in the fuzzing logs reported by Syzbot, whether or not the corresponding cases trigger bugs. Because all of them are resources for supplying us with valid caller-callee pairs. Consequently, we have collected a collection of 11,286 fuzzing logs, all of which were reported before the year 2022. Furthermore, we collected the indirect caller-callee pairs from them by leveraging MLTA, which has no false negatives. Any overlooked pair in this dataset would signify a false negative of GNNIC. Such results can be used to evaluate the false negatives of GNNIC conservatively. Notice that the real false-negative rate of GNNIC must be lower than the estimation provided by this method, due to the false positives of the MLTA. For instance, if a caller directly invokes a callee while a function pointer is dereferenced in this caller function, the MLTA might incorrectly identify the same callee as an indirect call target, when the function pointer and the callee share the same types. Our current method would classify such misidentifications as ground truth, increasing the estimated false negatives of GNNIC. Although these cases are relatively rare, as evidenced by our manual analysis.

We adopt this approach for several reasons. First, it allows us to gather sufficient data points, enhancing the statistical confidence of our analysis. Second, it helps minimize human bias in the evaluation of false negatives because humans often focus on confirming simpler true positives based on involved functions and pointer names while overlooking complex data flows. Third, we leverage open fuzzing logs, which offer comprehensive coverage, instead of relying solely on self-executed dynamic analysis. This approach overcomes limitations such as exploring a limited number of paths within a constrained time frame and the inability to access specific devices where certain execution results may be unattainable.

Method for false-positive evaluation. Most of the state-of-the-art works(e.g., ICallee [224], TypeDive [108], Crix [208]) only use the refining rate of indirect-call targets to show the effectiveness of their tools compared with the previous approaches. However, none of them has evaluated the real false-positive rate for the indirect-call

target identification, mainly due to the program complexity, which makes such evaluation time-consuming. To fill this gap and demonstrate the concrete performance of the current state-of-the-art, we manually performed a false-positive analysis. Specifically, we first sampled 100 indirect caller-callee pairs from the results of each following method: our work, type analysis, and type+multi-layer type analysis (MLTA). Then, we manually analyze such pairs to see if the target function can be a real target function or a false positive based on the following methods. ① If we can find at least one path that can pass the address of the target function to the call site, then we believe it is a true target function. ② If we cannot find such paths and the functionality of the target function is obviously different from the intention of the function pointer Then we believe it is a false target. Otherwise, we will consult as many auxiliary materials as possible, such as a fuzzing log, to determine whether the function is the real target function.

Data sampling for manual analysis. Static analysis techniques often rely on manual inspection of code segments or results to design or evaluate systems. Given the time-intensive and complex nature of the cases, manual analysis usually focuses on a sample of cases. When the total number of cases is substantial, state-of-the-art practices [225, 226, 227, 228, 208, 7, 198, 203] typically sample and manually analyze 40-400 cases. According to the previous study [229], such sampling ratios would result in a margin of error between ±5% to ±15%. In line with these state-of-the-art works and considering the complexity of manual analysis, we choose to select 100 - 300 samples for our case study (see §5.1.2) and false positive analysis respectively, ensuring that the margin of error falls within ±5% to ±10%, thereby providing statistically valid results. And to make sure the randomness of our sampled cases, we use pseudo-random number generators [230] to choose the data from the whole data set.

5.5.2 Scalability of GNNIC

GNNIC can analyze system-level programs in hours. For the Linux kernel and the Android kernel, which has more than 20 million lines of code, it takes about 10 minutes to collect all abstractive information from their corresponding LLVM IR. Furthermore, it takes about three hours to pre-train Word2Vec, build RAG, and train GNN with RAG. Notice that people do not need to re-train the Word2Vec model when analyzing different target programs because they are reusable. Also, benefiting from the inductive learning

of GraphSage, people do not need to re-train the whole GNN model after some new functions are added to the target program. At last, it takes about 10 minutes to generate the anchor functions and the final indirect call results. Thus, in total, it will take less than 4 hours to analyze the Linux or Android kernel. And the analysis time can be reduced to less than one hour for analyzing the FreeBSD kernel.

5.5.3 Evaluation on Anchor Function Identification

In this section, we assess the effectiveness of the anchor function identification process by examining instances of both false positives and false negatives.

The precision of the anchor functions. As discussed in §5.5.1, we have sampled and evaluated the precision of the 100 anchor functions. Specifically, in the 100 sampled indirect caller and target pairs, 94 of them are valid targets, and 6 are false positives. This result indicates that most of the anchor functions are valid targets. Moreover, after looking into the 6 cases, we find that all of them are caused by the implementation issue of name matching. Because on the source code level, GNNIC cannot perfectly catch the name of the function pointer and its container struct when meeting some complex code, such as nested complex macros. Such issues would require extensive engineering effort to resolve; however, they would not significantly enhance the overall system. Thus, we propose to regard them as potential future work.

Failures in anchor identification. Besides the precision, we also evaluate the failures in anchor identification where GNNIC cannot find any anchor functions for an indirect call. Specifically, for the Linux kernel, GNNIC fails to identify anchor functions for 6.3% of indirect calls. By manually looking into 50 failure cases, we found the following causes. ① 56% of them are caused by the complexity of source code, such as using complex Macros to initialize function pointers. ② 44% of these issues stem from function pointers that never directly take the target functions' address but instead acquire it from other function pointers. In §5.7, we will further discuss covering more anchor functions in the future.

5.5.4 The Precision Improvements on Target Identification

We compared GNNIC with the two-layer type analysis, which represents MLTA [108, 208], for the following reasons. First, to ensure a fair comparison of GNNIC, it is essential to evaluate it against source-code level or IR-level program analysis techniques, rather than binary-based approaches. Notably, recent methodologies, such as iCallee [224], specifically focus on binary code, which inherently lacks certain information when compared to IR-level analysis. Consequently, comparing GNNIC to such approaches will introduce biases that may potentially underestimate the contribution of their results. Second, based on the results of TypeDive [108], other IR or source-code level methods, such as pointer analysis, are not scalable for system-level programs and typically do not even surpass MLTA [108]. Third, using GNNIC with MLTA is scalable and can minimize false negatives, benefiting downstream applications.

In this evaluation, we tested the performance of GNNIC on the Linux kernel. As type analysis globally matches targets and finds a superset, we apply GNNIC on the results of type analysis to refine the target functions. Notice that, as we discussed in §5.1, not all the indirect calls qualify MLTA because some indirect calls only have one-layer type information. In this scenario, we step back to use the results of the one-layer type analysis.

The reduction rate of target functions. Figure 5.6 illustrates the proportion of target functions that can be reduced by the GNNIC system under varying threshold settings. In this graph, the x-axis represents the abstract similarity (the threshold) between the candidate target function and the anchor functions, as described in Section 5.3.4. The y-axis corresponds to the reduction rate of target functions at a specific threshold. As the abstract similarity threshold rises, the GNNIC system is capable of filtering a greater number of target functions, the majority of which are considered invalid. For example, for the Linux kernel, GNNIC can reduce about 88% of target functions when the abstract similarity is larger than 0.9.

Receiver operating characteristic (ROC) of GNNIC. To more effectively evaluate the performance of the GNNIC system, we examine the relationship between the false-positive rate (FPR = FP/(FP+TN)) and true-positive rate (i.e., recall, TPR = TP/TP+FN), reflecting the false negatives of GNNIC, in Figure 5.7. We used the

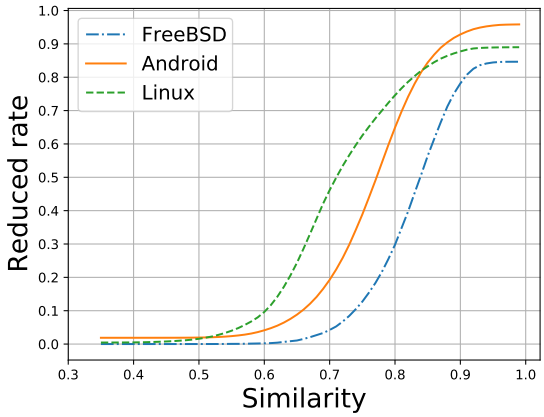

Figure 5.6: Percentage of refined indirect-call targets for different OS kernels.

3,831 ground-truth function pairs collected from fuzzing logs (see §5.5.1) to assess the true-positive rate evaluation of GNNIC. Furthermore, as mentioned in Section 5.5.1, we sampled and manually analyzed 100 indirect caller and callee pairs from the MLTA results. Our aim is to evaluate the false-positive rate of GNNIC based on these samples. To further minimize random error and to ensure the smoothness of the ROC curve, we further randomly sampled extra 200 pairs, bringing the total to 300 indirect caller and callee pairs from the MLTA to assist the false positive evaluation. Figure 5.7 demonstrates the effectiveness of GNNIC in achieving a desirable balance between false positive and true positive rates. At a false-positive rate of 0.33%, GNNIC achieves a true-positive rate (recall) of 84.8%, indicating 15.2% of false negatives. Under this condition, all the identified target functions are anchor functions. Conversely, when the true-positive rate (recall) reaches 99.6%, the corresponding false-positive rate rises to 60.7%. This underlines the precision of the GNNIC system in identifying the target functions of indirect calls.

Comparing with type-based approaches. A comparison between the results of GNNIC and type-based approaches against FPR can be problematic, as determining the total number of negative targets within the entire program for type analysis is challenging. This difficulty makes it hard to compute the false-positive rate. Nonetheless, we can still draw a comparison between the precision (precision = TP/(TP+FP)) and recall of GNNIC and type-based approaches. To this end, by manually checking all these 300 indirect caller and callee pairs, we can find 30 of these cases are true positives,

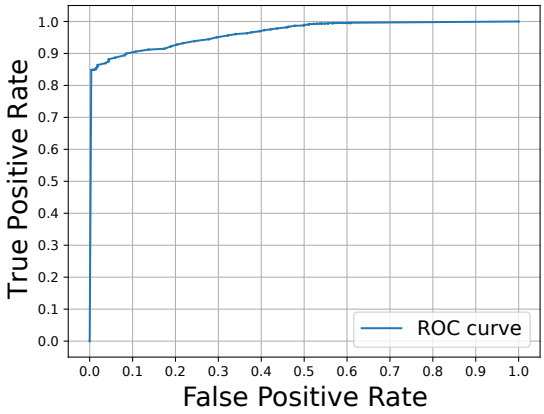

Figure 5.7: ROC curve for results for GNNIC on the Linux kernel.

which indicates that the precision of MLTA is 10%. Based on the mathematical proof of MLTA [108], in theory, there should be no false negatives, and thus the recall rate of MLTA is 100%. For GNNIC, when its precision reaches the highest value of 92.3%, the corresponding recall value is 84.8%. This precision aligns with the 94% precision of anchor functions, as discussed in Section 5.5.3. These findings suggest that GNNIC can improve precision by up to 82.3% compared to MLTA, while missing 16.2% of real targets.

5.5.5 Evaluating GNNIC on Different Projects

Besides the Linux kernel, we also evaluated GNNIC over the Android kernel and the FreeBSD kernel.

Distribution for the number of targets. We also evaluated the distribution for the number of indirect-call targets identified by the type-based approach and by GNNIC. Similar to the previous implementations, we reused the implementation of a type-based approach [208]. Table A.6 shows the distribution for the number of identified indirect call targets in the Linux kernel, Android kernel, and FreeBSD kernel. The percentage in the table indicates the percentage of indirect calls that have the corresponding number of target functions. The "Mean" and "Max" indicates the average and the maximum number of target functions for a specific indirect call. The results in Table A.6 show that for 7.2%, 7.2%, and 2.8% of indirect calls in the Linux kernel, Android kernel, and FreeBSD kernel, the type-based approach will detect more than 100 target functions for

them. However, GNNIC can reduce the corresponding number to 1.3%, 0.6%, and 0.2%, which shows that GNNIC can effectively refine the indirect call targets identified by the traditional type-based approaches.

Improvement for different kernels. Furthermore, Figure 5.6 also shows the percentage of reduced target functions based on the different similarity thresholds. Specifically, GNNIC can reduce 86% and 93% of target functions for the FreeBSD kernel and the Android kernel, which means GNNIC can effectively reduce the irrelevant target functions for different programs. However, due to missing enough ground truths for the FreeBSD and the Android kernel, we cannot draw the recall-precision curve for them.

5.6 Security Applications

5.6.1 Leveraging Abstract Similarity for Enhanced Bug Detection

We evaluate the effectiveness of GNNIC in real-world security applications by integrating it with static analysis tools for finding security bugs in the Linux and FreeBSD kernels. Our analysis uncovered a significant number of bugs, which we are currently reporting to the developers. To prioritize security, we will withhold the disclosure of these bugs until we have ensured they are thoroughly resolved.

Enhancing bug detection in similar functions through refined cross-checking techniques. Cross-checking techniques are widely utilized in static analysis for bug identification, as they collect similar code snippets and consider deviations as potential bugs [208, 62, 168]. However, these techniques face challenges, including requiring a large number of code snippets to establish correct usage patterns, cannot handle infrequent cases, and hard to identify deviations in certain situations. To address these limitations, we propose a combination approach to improve bug detection. We aim to identify overlooked bugs for uncommonly used functions, by combining our technique with cross-checking. We first cluster functions with abstract similarities and apply cross-checking to the entire class of functions, allowing for analysis of functions not commonly used and tolerating more noise. To illustrate its effectiveness, we use "allocation" as an example of abstract behavior and manually selected misused allocation functions as anchor functions, such as $mlx4\_alloc\_cmd\_mailbox$, from APISan [62] and Crix [208] results. Table A.7

shows that our analysis of the Linux and FreeBSD kernels found 97 missing return value check bugs for 38 different allocation functions, with 63% (24/38) of those functions used less than 10 times in the program. All of these instances have the potential to trigger NULL-pointer dereference problems when the unchecked return pointers are further dereferenced. Traditional cross-checking techniques in Crix and APISan failed to report these bugs due to their limitations in analyzing infrequently used or misused functions with a higher deviation proportion than the threshold. Such results demonstrate that combining our method with cross-checking can effectively identify more code issues, by reducing the false negatives of the original cross-checking-based techniques. On the other hand, GNNIC can also help reduce the false positives of traditional static analysis by providing them with more precise call graphs. A more in-depth discourse on this aspect will be presented later.

Interesting findings during the bug detection. Incorporating abstract similarity of functions with static analysis not only enhances the capabilities of cross-checking techniques but also reveals new bugs that conventional cross-checking methods would completely miss. This is because if a function is used in a single, uniform pattern, the cross-checking technique cannot detect any issues. For instance, if the return value of a function is never checked throughout the entire program, the cross-checking approaches, such as Crix, will not identify the missing check issue. Similarly, if most of the usages (>=50%) of a function are incorrect, cross-checking methods would also miss them. However, such bugs can be identified by integrating abstract-similarity-based clustering. Due to such reasons, 33 out of 97 missing NULL-check bugs against 10 allocation functions would be totally missed by cross-checking-only approaches focusing on security checks. More interestingly, the return values from 3 of these allocation functions were never checked by security checks and thus will have gone unnoticed using only cross-checking techniques. Therefore, combining abstract similarity and static program analysis dramatically increases the ability to find new and previously overlooked bugs in code.

Reducing false positives in static analysis. Static analysis tools are widely used in both industry and academia to enhance software security and quality. Indirect-call analysis is critical for these tools, but false positives can impede bug detection. For example, as the claims from INCRELUX [198] and DiffCVSS [203], using traditional techniques, such as type analysis, to identify indirect calls will introduce high false

positives into their results and sometimes even make the tool unusable. Our study found that imprecise indirect call analysis is a major contributor to the results of Crix, accounting for 59% of the total warnings. Integrating our system with Crix reduced these cases by 74%, resulting in a 44% reduction in total false positives. This demonstrates the potential of our system to deliver high-accuracy results when combined with conventional program analysis techniques.

5.6.2 Other Potential Security-related Applications of GNNIC

Enhancing vulnerability-reachability analysis. Another potential security application of the GNNIC is assessing the reachability of a vulnerability [231]. An analysis of call chains from Syzbot [223] reveals that, on average, there are 4.7 indirect calls in a chain within the Linux kernel. Also, based on the average number of target functions identified by GNNIC and type-based approach (see §5.5.5), we can infer that in order to find the correct call chain; we need to analyze $84.7^{4.7}$ (10^9) candidate call chains based on the call graph constructed by the traditional approaches. However, with the accurate call graph built by GNNIC, we only need to analyze $9.7^{4.7}$ (10^4) candidate call chains. This result indicates that the call graph built with GNNIC can dramatically improve the precision of call chain identification, thereby supporting future security-related research, such as vulnerability assessment.

Expanding bug identification capabilities with abstract similarity. In §5.6.1, we demonstrated the potential of abstract similarity in enhancing bug identification. Our example of missing NULL checks for allocation-related functions showcases the effectiveness of our approach. However, it is essential to recognize that the application of abstract similarity is not confined to this specific type of bug and can be employed to detect other common bugs as well. Future research could concentrate on investigating the use of abstract similarity for identifying various bug types, such as permission issues in critical APIs. As highlighted in PEX [168], abstract similarity holds the potential for aiding in the detection of similar permission issues. Although this kind of research is time-consuming and beyond the primary scope of this paper, it offers exciting prospects for further exploration.

Improving directed fuzzing and concolic execution. Directed fuzzing and concolic

execution often require knowledge of indirect-call targets during the execution. However, due to accuracy limitations or performance issues or existing approaches, only a few existing tools effectively handle indirect calls, such as HFL [212] and CollAFL [207]. Conversely, many dynamic analysis tools and concolic-execution tools, such as previous works [232, 205, 206, 88, 233, 49, 234, 235, 236, 48], do not handle indirect branches at all. However, neglecting to correctly handle indirect calls result in dynamic analysis tools missing many targets. For example, Böhme et al. [201] proposed a Greybox Fuzzing (DGF) based on the AFLGo, which found that *"more than half of the changed basic blocks are accessible only via register indirect calls or jumps (e.g., from function-pointers). Those do not appear as edges in the analyzed call-graph or in the control flow graph."* These findings suggest that over-approximated approaches, like type-based analysis, may generate numerous false positives, affecting downstream application accuracy and increasing analysis time. On the other hand, under-approximated approaches may introduce false negatives in downstream applications. Consequently, an accurate call-graph construction tool like GNNIC holds significant potential for enhancing existing directed fuzzing and concolic execution instruments. Furthermore, users can optimize the balance between false positives and false negatives by adjusting the thresholds.

5.7 Discussion

Comparison with type-based indirect-call analysis. Although type-based indirect call analysis can generate many false positives in large software, in theory, it can ensure soundness. This is essential for some special applications, such as control-flow integrity (CFI). However, many security applications do not require soundness but accuracy (balanced precision and recall). Most recently, Lu proposed the Range-Aware Multi-layer Type Analysis (MLTA) [237] in order to enhance the accuracy of indirect call target identification. Although this method holds the potential to improve the performance of security applications, our study (see §5.1) and evaluation results (see §5.5) have shown that, due to the prevalence of cases where the scope of type-based approaches is exceeded, this method still falls short in providing precise assistance for program static analysis, such as inter-procedural taint analysis. More importantly, our system, which is designed to filter target functions based on abstract similarity, can provide complementary benefits

to these type-based approaches, including the range-aware MLTA. By incorporating the strengths of both approaches, the performance of both our and their system can be further improved.

Comparison with CFI-based indirect-call analysis. In the realm of refining indirect call targets, several state-of-the-art approaches, including OS-CFI [210], τCFI [238], and PathArmor [239], utilize CFI techniques to refine indirect call targets during runtime. While these approaches benefit from dynamic analysis and can effectively analyze user space programs with high precision, they suffer from low coverage rates for large programs, particularly OS-level software, and cannot be easily applied to low-level programs, such as device drivers, without hardware support. For example, HFL [212] and DR.FUZZ [213], which are two recent kernel fuzzers, show that the code coverage for Linux kernel and its drivers is typically less than 10%, and sometimes even less than 1% for some drivers. Beyond this, as discussed in OS-CFI, such CFI-based techniques typically require hardware features, including Intel MPX, Intel TSX, and Intel PT. But, in contrast, our approach is based on static code analysis and does not require hardware support, making it suitable for analyzing large and low-level programs. Thus, while both our approach and CFI-based techniques aim to identify indirect call targets, they cater to different types of programs and have distinct security applications.

Path-based indirect-call analysis. Most of the existing indirect-call detection approaches aim to give all the possible target functions for a given indirect call. However, for some situations that require analyzing the specific execution paths, such as path-sensitive analysis, the target functions given by the current approaches are too general. For example, although, on average, we can reduce the number of cases for finding the correct call chain from 10^9 to 10^4, based on our results compared to type-based methods (see §5.6.1), this is still not unique. The target function of the indirect call should be uniquely identifiable by giving enough preconditions and constraints. However, such path-sensitive indirect call analysis is out of this project's scope. We would like to regard it as interesting future work.

Limitation of current work and potential future works. First, finding the anchor functions is an unexplored problem, and none of the existing methods can handle this problem well. So in this project, we chose to combine the IR level information with the source code level information to find the anchor functions within the restricted scope.

However, due to the lack of accurate matching methods from IR variables to variables on the source code, and the lack of accurate source code or pre-compiled code analysis tools, we cannot match the name of each function pointer with one hundred percent of accuracy. This part of the problem can be alleviated in the future by improving the precision of source-level value and name analysis. Such a tool is preferably compiler-based, which can make the analysis more accurate. In this way, it automatically extracts the names and initializers of all function pointers during the compilation process, thereby assisting GNNIC in associating function pointers with target functions.

5.8 Related Work

Refining indirect-call targets. Recently a number of works on refining indirect-call targets emerge, including value-set analysis, point-to analysis, type analysis, and Neural Networks. Some previous works are working on refining the indirect-call targets on the binary level. For example, BPA [209] is a binary-level points-to analysis framework based on a block memory model, which can improve precision by 34.5% compared with other binary-level techniques. Icallee [224] is a Siamese Neural Network approach that uses NLP to embed the context of call instructions, reducing indirect call targets compared to other binary-level approaches like BPA and τCFI [238] but has better performance. However, such approaches are only tested on small programs, such as libraries, and are not scalable on large programs, such as OS kernels. Also, currently, the source-level solutions can still generate much better performance than the binary-level approaches, such as Icallee. Source-code level and IR-level pointer-based approaches, such as K-miner [240] and SVF [241], are also used to analyze function pointers. However, as we discussed, such approaches are typically not scalable for large programs or have a high false-negative or false-positive rate. Therefore, as we discussed, type-based approaches are typically the best choice for source-code level indirect-call targets identification. And moreover, multi-layer type-based solutions [208, 108] have been proposed in recent years, which improve the type-based approaches and have much better results. However, still, many function pointers do not qualify multi-layer type analysis, and our evaluation (see §5.5) shows that for kernel-level large programs, GNNIC can still improve such state-of-the-art approaches a lot.

Measuring the similarity of code. Machine learning techniques are widely used in program and code similarity analysis. Code2Vec [242] can embed code pieces into fixed-length code vectors by aggregating the information of collected code paths in the abstract syntax tree (AST). Func2Vec [68] is based on a neural network, which can generate function vectors by random walking on the interprocedural control-flow graph of programs. FuncGNN [243] is a graph neural network trained on a labeled control-flow graph, which can be used to estimate the graph edit distance of the program. FA-AST [244] trains the GNN on a flow-augmented AST, which can catch the data and control-flow information of the program. Gemini [194] is a neural network-based approach to compute the embedding of functions based on their control flow graph at the binary level. All the above techniques are based on the program's control and data flow, which can be used in clone detection or predict semantic properties of the code snippet. Unlike these works, some machine learning models are also trained on program call graphs. Xu et al. [245] can detect Android malware, which leverages the natural language processing techniques to compute the embedding of the program by analyzing the whole call graph. DeepCatra [246] is designed to detect the malware behaviors of Android APPs. DeepCatra consists of a bidirectional LSTM (BiLSTM) and a GNN, which is trained on the features from the call trace of the program. Different from all these works, GNNIC focuses on the function call and types related information, representing the features of indirect-call target functions better.

Chapter 6

Conclusion

This thesis provides a comprehensive summarization of our past and current works, which leverage code semantics to strength program security and enhance program analysis. Specifically, we draw the following conclusions from our works.

6.1 Conclusion of HERO

Large programs such as OS kernels usually have complicated error-handling and code-cleanup mechanisms, which are buggy because they are less tested and hard to implement. Prior research attempted to detect the bugs, but mainly on the "handling" part instead of the cleanup mechanisms. This paper proposed DiEH bugs, a class of error-handling bugs that are caused by improper cleanup operations—incorrect-order, redundant, and inadequate cleanups. Through a study, we show that DiEH is hard to avoid and thus is prevalent; it also causes critical security problems such as memory corruption and privilege escalation. This paper then presented a new detection system, HERO. At its core is a precise function pairing technique that leverages the unique error-handling structures in low-level languages. We evaluate HERO on two OS kernels and the OpenSSL library. The results show that HERO can precisely identify a large number of function pairs, including custom ones, and can detect 239 critical DiEH bugs, most of which were confirmed by maintainers. HERO is generic, and its precise pairing analysis can be applied to benefit other research, such as race detection and temporal-rule inferences.

6.2 Conclusion of SID

Maintainers of large software programs are bombarded with a large number of bug reports without a reliable description of security impacts. With limited resources, maintainers have to prioritize the patching for bugs with security impacts, which is, however challenging, and de-prioritizing a security-critical bug will lead to critical security problems. This paper presented SID, an automated approach to determining the security impacts for a massive number of bug patches. The core of SID is the symbolic rule comparison mechanism that employs differential, under-constrained symbolic execution to precisely confirm the security impacts of a bug. SID can further automatically classify vulnerabilities based on their security impacts. We have implemented SID and applied it to determine the security impacts of Linux-kernel bugs. As a result, SID has found 227 SID from 54K valid commits patches in the Linux kernel, and 21 of them remain unpatched in the latest Android kernel (version 4.14), which may cause critical security problems for Android devices. Many of the identified security bugs have been assigned with a CVE ID and a high CVSS score. The evaluation results show the precision and effectiveness of SID in automatically determining security impacts.

6.3 Conclusion of DiffCVSS

CVSS is used in an "one for all" strategy that assigns a single severity score, regardless of the derivatives or versions. This problem results in both severity overestimation, which wastes maintenance resources and severity underestimation, which delays the patching of critical vulnerabilities and incurs critical threats. To address it, this paper presents DiffCVSS, a system that can automatically and precisely determine if a vulnerability will have a higher or lower severity in a different OS. DiffCVSS incorporates multiple new techniques, such as automatically identifying the call-chain for a vulnerability and mapping kernel functions to CVSS metrics, to ensure precision and effectiveness. We evaluated DiffCVSS on the Linux and Android kernels. DiffCVSS reveals that 110 (86.7%) of vulnerabilities show a different severity across OSes, and thus should be reevaluated per OS. In 18 cases, the severity is higher in the derivative Android system; failure to re-assess them will delay the patching process, incurring critical threats. Our user study

also showed that DiffCVSS can correctly and effectively guide OS-aware re-evaluation. The results confirm that DiffCVSS is precise and effective in capturing severity differences across OSes.

6.4 Conclusion of GNNIC

In this paper, we introduce GNNIC, an innovative call graph construction tool that seamlessly combines program analysis and GNN to accurately identify indirect-call targets. This enables the creation of highly precise call graphs for large programs. GNNIC uses abstract-similarity analysis to match target functions based on the abstract behaviors they share. To realize this approach, we propose techniques, such as scoped unique-name matching, to identify anchor functions, and the use of a *representative abstraction graph* to incorporate diverse information about a function. The graph can be fed to GNN for training embedding models. Our evaluation results show that GNNIC significantly improves precision compared to existing state-of-the-art techniques. Additionally, we demonstrate that using function abstraction-based similarity analysis alongside precise call graphs can be effective in a wide range of security applications.

References

[1] Huqiu Liu, Yuping Wang, Lingbo Jiang, and Shimin Hu. Pf-miner: A new paired functions mining method for android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 33–42. IEEE, 2014.

[2] Hu-Qiu Liu, Jia-Ju Bai, Yu-Ping Wang, Zhe Bian, and Shi-Min Hu. Pairminer: mining for paired functions in kernel extensions. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 93–101. IEEE, 2015.

[3] The android for msm project, 2021. `https://source.codeaurora.org/quic/la/kernel/msm-3.10/`.

[4] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2041–2058, 2021.

[5] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, 2019.

[6] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically identifying security checks for detecting kernel semantic bugs. In *European Symposium on Research in Computer Security*, pages 3–25. Springer, 2019.

[7] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[8] Qiushi Wu, Yue Xiao, Xiaojing Liao, and Kangjie Lu. OS-Aware vulnerability prioritization via differential severity analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 395–412, Boston, MA, August 2022. USENIX Association.

[9] Qiushi Wu, Zhongshu Gu, Hani Jamjoom, and Kangjie Lu. Gnnic: Finding long-lost sibling functions with abstract similarity. In *NDSS*, 2024.

[10] Cvss calculator, 2020. `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

[11] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[12] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: finding refcount bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 531–544, 2016.

[13] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[14] MITRE-CVE. Cvedetils, 2020. `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html`.

[15] MITRE Corporation. Cwe-416: Use after free, 2020. `https://cwe.mitre.org/data/definitions/416.html`.

[16] MITRE Corporation. Cwe-200: Exposure of sensitive information to an unauthorized actor, 2020. `https://cwe.mitre.org/data/definitions/200.html`.

[17] MITRE Corporation. Owasp top ten 2004 category a9 - denial of service, 2020. https://cwe.mitre.org/data/definitions/730.html.

[18] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1663–1674, 2016.

[19] Harold A Rosenberg and Kang G Shin. Software fault injection and its application in distributed systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 208–217. IEEE, 1993.

[20] Cindy Rubio-González, Haryadi S Gunawi, Ben Liblit, Remzi H Arpaci-Dusseau, and Andrea C Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM Sigplan Notices*, volume 44, pages 270–280. ACM, 2009.

[21] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 472–482. ACM, 2016.

[22] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762. ACM, 2017.

[23] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

[24] Rusty Russell. What if I don't actually like my users?, April 2008. https://ozlabs.org/~rusty/index.cgi/tech/2008-04-01.html.

[25] MITRE-CVE. A double-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15504/.

[26] MITRE-CVE. A use-after-free in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15292/.

[27] MITRE-CVE. A null dereference vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15923/.

[28] MITRE-CVE. A refcount leak vulnerability in the freebsd, 2019. https://www.cvedetails.com/cve/CVE-2019-5607/.

[29] MITRE-CVE. A refcount leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2016-0728/.

[30] Corey Minyard and Thomas Hellstrom. Cve-2019-0685: A refcount leak vulnerability., 2004. https://sigpwn.io/blog/2020/5/7/cve-2019-0685-win32k-reference-count-leak.

[31] MITRE-CVE. Cve-2019-0685, 2020. ttps://www.cvedetails.com/cve/CVE-2019-0685/.

[32] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-16994/.

[33] MITRE-CVE. A deadlock vulnerability in the linux kernel, 2019. https://www.cvedetails.com/cve/CVE-2019-15538/.

[34] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82. IEEE, 2019.

[35] Sen Zhang, Jingwen Zhu, Ao Liu, Weijing Wang, Chenkai Guo, and Jing Xu. A novel memory leak classification for evaluating the applicability of static analysis tools. In *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 351–356. IEEE, 2018.

[36] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786. USENIX Association, 2019.

[37] Bootlin-Community. Linux kernel: kobject_init_and_add(), 2020. https://elixir.bootlin.com/linux/v5.7-rc7/source/lib/kobject.c#L464.

[38] LLVM project community. Llvm alias analysis infrastructure, 2020. https://llvm.org/docs/AliasAnalysis.html.

[39] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[40] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 43–52. IEEE, 2009.

[41] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[42] Bjorn Andersson. Linux kernel patch log, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6e5da6f7d82474e94c2d4a38cf9ca4edbb3e03a0.

[43] MITRE Corporation. Common weakness enumeration (cwe), 2020. https://cwe.mitre.org/.

[44] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15393.

[45] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8980.

[46] MITRE-CVE. A memory leak vulnerability in the linux kernel, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5023.

[47] Thgarnie. Syzkaller, 2019. https://github.com/google/syzkaller.

[48] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for

kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[49] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[50] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[51] 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.

[52] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[53] Jia-Ju Bai, Hu-Qiu Liu, Yu-Ping Wang, and Shi-Min Hu. Runtime checking for paired functions in device drivers. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 407–414. IEEE, 2014.

[54] Suman Saha, Julia Lawall, and Gilles Muller. An approach to improving the structure of error-handling code in the linux kernel. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 41–50, 2011.

[55] Wensheng Tang. Identifying error code misuses in complex system. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 428–432, 2019.

[56] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V Thakur. Effective error-specification inference via domain-knowledge expansion. In

Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 466–476, 2019.

[57] Aditya Pakki and Kangjie Lu. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *27th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2020.

[58] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*, pages 496–506. IEEE, 2009.

[59] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in openssl using coccinelle. In *2010 European Dependable Computing Conference*, pages 191–196. IEEE, 2010.

[60] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[61] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[62] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing {API} usages through semantic cross-checking. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 363–378, 2016.

[63] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, 2007.

[64] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 119–130, 2010.

[65] Westley Weimer and George C Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[66] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54, 2017.

[67] Pan Bian, Bin Liang, Yan Zhang, Chaoqun Yang, Wenchang Shi, and Yan Cai. Detecting bugs by discovering expectations and their violations. *IEEE Transactions on Software Engineering*, 45(10):984–1001, 2018.

[68] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779*, 2018.

[69] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically identifying security checks for detecting kernel semantic bugs. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 3–25, Cham, 2019. Springer International Publishing.

[70] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX*, pages 35–39, October 2005.

[71] Mozilla. Bugzilla main page, 2019. `https://bugzilla.mozilla.org/home`.

[72] Linus Torvalds. Linux kernel source tree, 2019. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/`.

[73] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[74] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 518–521. IEEE, 2015.

[75] Jeff Arnold, Tim Abbott, Waseem Daher, Gregory Price, Nelson Elhage, Geoffrey Thomas, and Anders Kaseorg. Security impact ratings considered harmful. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems*, May 2009. https://www.usenix.org/conference/hotos-xii/security-impact-ratings-considered-harmful.

[76] Christine Hall. Survey shows linux the top operating system for internet of things devices, 2018. https://www.itprotoday.com/iot/survey-shows-linux-top-operating-system-internet-things-devices.

[77] Stat Counte. Mobile operating system market share worldwide, 2019. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[78] Google. Android security rewards program rules, 2019. https://www.google.com/about/appsecurity/android-rewards/.

[79] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. Vulnerability identification and classification via text mining bug databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, pages 3612–3618. IEEE, 2014.

[80] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

[81] Dumidu Wijayasekara, Milos Manic, Jason L Wright, and Miles McQueen. Mining bug databases for unidentified software vulnerabilities. In *2012 5th International Conference on Human System Interactions*, pages 89–96. IEEE, 2012.

[82] Jacob P Tyo. *Empirical analysis and automated classification of security bug reports.* West Virginia University, 2016.

[83] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919. ACM, 2017.

[84] Diksha Behl, Sahil Handa, and Anuja Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pages 294–299. IEEE, 2014.

[85] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. *The 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, 2010.

[86] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407. ACM, 2017.

[87] Yuan Tian, Nasir Ali, David Lo, and Ahmed E Hassan. On the unreliability of bug severity data. *Empirical Software Engineering*, 21(6):2298–2323, 2016.

[88] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[89] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[90] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. Automatically generating malicious disks using symbolic execution. In *2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 243–257, May 2006.

[91] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, pages 279–292, October 2012.

[92] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913. ACM, 2018.

[93] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[94] 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.

[95] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[96] 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.

[97] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

[98] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. IEEE, 2016.

[99] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[100] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: a permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[101] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377. ACM, 2015.

[102] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.

[103] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[104] A Yu Gerasimov, Leonid V Kruglov, MK Ermakov, and Sergey P Vartanov. An approach to reachability determination for static analysis defects with the help of dynamic symbolic execution. *Programming and Computer Software*, 44(6):467–475, 2018.

[105] Linux syscall reference, 2019. `https://syscalls.kernelgrok.com/`.

[106] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Fuzzer for linux kernel drivers, 2019. `https://github.com/ucsb-seclab/difuze`.

[107] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[108] 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.

[109] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM, 2017.

[110] MITRE Corporation. Common vulnerabilities and exposures, 2019. `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+kernel`.

[111] nluedtke. Linux kernel cves, github, 2019. `https://github.com/nluedtke/linux_kernel_cves`.

[112] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 226–237, 2016.

[113] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 409–423, 2014.

[114] Linus Torvalds. Linux kernel 4.14, 2017. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v4.14`.

[115] Aravind Machiry, Nilo Redini, E Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[116] Zerong Zhang, Yongyan Wang, and Zhimin Fan. Similarity analysis between scale model and prototype of large vibrating screen. *Shock and Vibration*, 2015, 2015.

[117] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *ACM SIGPLAN Notices*, volume 36, pages 47–58. ACM, 2001.

[118] Dipok Chandra Das and Md Rayhanur Rahman. Security and performance bug reports identification with class-imbalance sampling and feature selection. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 316–321. IEEE, 2018.

[119] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.

[120] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[121] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[122] Nuthan Munaiah and Andrew Meneely. Vulnerability severity scoring and bounties: why the disconnect? In *Proceedings of the 2nd International Workshop on Software Analytics*, pages 8–14. ACM, 2016.

[123] Tim Menzies and Andrian Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355. IEEE, 2008.

[124] KK Chaturvedi and VB Singh. Determining bug severity using machine learning techniques. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–6. IEEE, 2012.

[125] Nivir Kanti Singha Roy and Bruno Rossi. Towards an improvement of bug severity classification. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 269–276. IEEE, 2014.

[126] Waheed Yousuf Ramay, Qasim Umer, Xu Cheng Yin, Chao Zhu, and Inam Illahi. Deep neural network-based severity prediction of bug reports. *IEEE Access*, 7:46846–46857, 2019.

[127] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[128] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[129] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–4, July 2007.

[130] Wikipedia contributors. Mobile operating system — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-October-2021].

[131] Google. Android developers, 2021. `https://developer.android.com/`.

[132] Security updates and resources, 2021. `https://source.android.com/security/overview/updates-resources`.

[133] Incident Response and Security Teams. Cvss score, 2021.

[134] MITRE-CVE. Common vulnerabilities and exposures program, 2021.

[135] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International symposium on empirical software engineering and measurement*, pages 535–544. IEEE, 2009.

[136] RedHat. Third-party severity ratings, 2007. `https://www.redhat.com/en/blog/third-party-severity-ratings`.

[137] Ubuntu Security Team. Ubuntu cve tracker. `https://git.launchpad.net/ubuntu-cve-tracker/tree/README#n229`.

[138] BlackBerry Limited. Blackberry secure platform, 2018. `https://www.blackberry.com/us/en/campaigns/2018/blackberry-secure-platform`.

[139] Luca Allodi, Sebastian Banescu, Henning Femmer, and Kristian Beckers. Identifying relevant information cues for vulnerability assessment using cvss. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 119–126, 2018.

[140] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927, 2018.

[141] Tao Chen, Ruifeng Xu, Yulan He, and Xuan Wang. Improving sentiment analysis via sentence type classification using bilstm-crf and cnn. *Expert Systems with Applications*, 72:221–230, 2017.

[142] Google. Android common kernels, 2020. `https://source.android.com/devices/architecture/kernel/android-common`.

[143] OPPO. Color os7, 2020. `https://www.coloros.com/en/coloros7`.

[144] Ltd Huawei Device Co. Emui 11, 2021. `https://consumer.huawei.com/en/emui-11/`.

[145] MIUI. Miui 12, 2020. `https://en.miui.com/`.

[146] Google. Chrome os, 2020. `https://www.google.com/chromebook/chrome-os/`.

[147] Android. Security updates and resources, 2020. `https://www.redhat.com/en/blog/third-party-severity-ratings`.

[148] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, Austin, TX, August 2016. USENIX Association.

[149] NVD ITL. National vulnerability database, 2020. `https://nvd.nist.gov/`.

[150] Tenable Research. Predictive prioritization: How to focus on the vulnerabilities that matter most, 2018.

[151] 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)*, 2021.

[152] Wiseman, Greg. How to measurably reduce false positive vulnerabilities by up to 22%., 2020.

[153] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 535–544, 2009.

[154] The kernel development community. Writing kernel-doc comments, 2021.

[155] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, 2008.

[156] The kernel development community. The linux kernel api.

[157] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[158] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1555–1565, 2014.

[159] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting linux for mobile platforms: An empirical study of android. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 629–632. IEEE, 2012.

[160] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Ndss*, volume 310, pages 20–38, 2013.

[161] Eugene Charniak. Tree-bank grammars. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1031–1036, 1996.

[162] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, Adriane Boyd, et al. spacy: Industrial-strength natural language processing in python, 2020.

[163] Kevin Bartz, Jack W Stokes, John Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihile. Finding similar failures using callstack similarity. In *SysML08: Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2008.

[164] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[165] Said Sryheni. Find all simple paths between two vertices in a graph, 2020. `https://www.baeldung.com/cs/simple-paths-between-two-vertices`.

[166] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[167] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[168] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[169] Diffcvss, 2021. `https://sites.google.com/view/diffcvss/home`.

[170] FIRST. A complete guide to the common vulnerability scoring system, 2021.

[171] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[172] Steven S Muchnick. Advanced compiler design and implementation morgan kaufmann publishers. *San Francisco, California*, 1997.

[173] National Vulnerability Database (NVD). Vulnerability metrics, 2020. `https://nvd.nist.gov/vuln-metrics/cvss`.

[174] Debjani Saha, Anna Chan, Brook Stacy, Kiran Javkar, Sushant Patkar, and Michelle L Mazurek. User attitudes on direct-to-consumer genetic testing. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 120–138. IEEE, 2020.

[175] Miranda Wei, Madison Stamos, Sophie Veys, Nathan Reitinger, Justin Goodman, Margot Herman, Dorota Filipczuk, Ben Weinshel, Michelle L Mazurek, and Blase Ur. What twitter knows: Characterizing ad targeting practices, user perceptions, and ad explanations through users' own twitter data. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 145–162, 2020.

[176] Allison McDonald, Catherine Barwulor, Michelle L Mazurek, Florian Schaub, and Elissa M Redmiles. " it's stressful having all these phones": Investigating sex workers' safety goals, risks, and practices online. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[177] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[178] Daniel Votipka, Eric Zhang, and Michelle L Mazurek. Hacked: A pedagogical analysis of online vulnerability discovery exercises. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1268–1285. IEEE, 2021.

[179] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1875–1892, 2020.

[180] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 597–616, 2021.

[181] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 621–637, 2018.

[182] Atefeh Khazaei, Mohammad Ghasemzadeh, and Vali Derhami. An automatic method for cvss score prediction using vulnerabilities description. *Journal of Intelligent & Fuzzy Systems*, 30(1):89–96, 2016.

[183] 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.

[184] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 125–136. IEEE, 2017.

[185] Andrej Dobrovoljc, Denis Trček, and Borut Likar. Predicting exploitations of information systems vulnerabilities through attackers' characteristics. *IEEE Access*, 5:26063–26075, 2017.

[186] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114, 2010.

[187] Benjamin L Bullough, Anna K Yanchenko, Christopher L Smith, and Joseph R Zipkin. Predicting exploitation of disclosed software vulnerabilities using open-source data. In *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, pages 45–53, 2017.

[188] Qixu Liu, Yuqing Zhang, Ying Kong, and Qianru Wu. Improving vrss-based vulnerability prioritization using analytic hierarchy process. *Journal of Systems and Software*, 85(8):1699–1708, 2012.

[189] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. Assessment of vulnerability severity using text mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, pages 1–6, 2017.

[190] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[191] Katheryn A Farris, Ankit Shah, George Cybenko, Rajesh Ganesan, and Sushil Jajodia. Vulcon: A system for vulnerability prioritization, mitigation, and management. *ACM Transactions on Privacy and Security (TOPS)*, 21(4):1–28, 2018.

[192] Ruchi Sharma, Ritu Sibal, and Sangeeta Sabharwal. Software vulnerability prioritization using vulnerability description. *International Journal of System Assurance Engineering and Management*, pages 1–7, 2020.

[193] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 896–899. IEEE, 2018.

[194] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[195] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[196] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[197] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, pages 3063–3069, 2020.

[198] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *2022 Network and Distributed System Security Symposium*, 2022.

[199] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[200] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[201] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[202] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[203] Qiushi Wu, Yue Xiao, Xiaojing Liao, and Kangjie Lu. Os-aware vulnerability prioritization via differential severity analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 395–412, 2022.

[204] Gogul Balakrishnan, T Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, S Yong, C-H Chen, and Tim Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In *International Conference on Computer Aided Verification*, pages 158–163. Springer, 2005.

[205] Vincent M Weaver and Dave Jones. perf fuzzer: Targeted fuzzing of the perf event open () system call. *UMaine VMW Group, Tech. Rep*, 2015.

[206] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *USENIX Security Symposium*, pages 149–165, 2017.

[207] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[208] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[209] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium, NDSS*, 2021.

[210] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *USENIX Security Symposium*, pages 195–211, 2019.

[211] Ca Technologies Inc. Dynamic call tracking method based on cpu interrupt instructions to improve disassembly quality of indirect calls, 2020.

[212] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[213] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *NDSS*, 2022.

[214] CSIRO's Data61. Stellargraph machine learning library. `https://github.com/stellargraph/stellargraph`, 2018.

[215] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[216] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, Vancouver, BC, August 2017. USENIX Association.

[217] Changming Liu, Yaohui Chen, and Long Lu. Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[218] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[219] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, 2012.

[220] S Vijayarani, Ms J Ilamathi, Ms Nithya, et al. Preprocessing techniques for text mining-an overview. *International Journal of Computer Science & Communication Networks*, 5(1):7–16, 2015.

[221] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[222] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[223] Syzcaller group. Syzbot fuzzing logs, 2022.

[224] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Chao Zhang, Zhijian Ou, and Min Yang. icallee: Recovering call graphs for binaries. *arXiv preprint arXiv:2111.01415*, 2021.

[225] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[226] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 613–624, 2019.

[227] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020.

[228] Lirong Fu, Shouling Ji, Kangjie Lu, Peiyu Liu, Xuhong Zhang, Yuxuan Duan, Zihui Zhang, Wenzhi Chen, and Yanjun Wu. Cpscan: Detecting bugs caused by code pruning in iot kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 794–810, 2021.

[229] Ronán Conroy. Sample size: A rough guide. *Retreived from http://www. beaumontethics. ie/docs/application/samplesizecalculation. pdf*, 2015.

[230] Python Software Foundation. random — generate pseudo-random numbers, 2023.

[231] Awad A Younis, Yashwant K Malaiya, and Indrajit Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8. IEEE, 2014.

[232] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for cots operating systems. In *USENIX Annual Technical Conference*, pages 689–701, 2017.

[233] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[234] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[235] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[236] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *USENIX Security Symposium*, pages 729–743, 2018.

[237] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1610–1624. IEEE Computer Society, 2023.

[238] Jens Grossklags and Claudia Eckert. τcfi: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050, page 423. Springer, 2018.

[239] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940, 2015.

[240] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[241] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[242] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[243] Aravind Nair, Avijit Roy, and Karl Meinke. Funcgnn: a graph neural network approach to program similarity. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[244] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020.

[245] Peng Xu, Claudia Eckert, and Apostolis Zarras. Detecting and categorizing android malware with graph neural networks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 409–412, 2021.

[246] Yafei Wu, Jian Shi, Peicheng Wang, Dongrui Zeng, and Cong Sun. Deepcatra: Learning flow-and graph-based behaviors for android malware detection. *arXiv preprint arXiv:2201.12876*, 2022.

Appendix A

Appendix: Long Table and Other Data

User study materials: Recruitment Requirement for Students

Students met the following requirement are selected to participate the user study.

- Must have take at least one security-related class
- Must have some basic knowledge of system security
- Must read/write vulnerability report before
- Must be familiar with Linux Kernel

Contact Information Survey

The contact information survey is used to record demographics information and contact method, provide the online consent form. The details can be found at [169].

Online Consent Form

We provide online consent form for participants to read before agreeing to be in the study. It includes the purpose, the procedure of this study, and the risks and benefits of taking part in this study. The details can be found at [169]

Train Evaluation Form

Train Evaluation Form, which can evaluate the student's professional knowledge of vulnerability assessment, is used to guarantee the student participants can deliver qualified responses. The details can be found at [169].

Post Survey

1. How necessary do you think the derivatives of Linux kernel need to re-evaluate the vulnerabilities?
   (a) Extremely necessary
   (b) Very necessary
   (c) Moderately necessary
   (d) Slightly necessary
   (e) Not necessary at all

2. Do you think DiffCVSS can correctly guide you when re-evaluate the vulnerability?
   ☐ Yes
   ☐ No
   ☐ Not Sure

3. How effectively do you think DiffCVSS tool can help you re-evaluate vulnerability?
   (a) Extremely effective
   (b) Very effective
   (c) Moderately effective
   (d) Slightly effective
   (e) Not effective at all

4. What kinds of manual work can be reduced with DiffCVSS?
   ☐ Less time to find vulnerability-related call-chain
   ☐ Less time to understand the functionality of code
   ☐ Less time to analyze the CVSS metrics
   ☐ Less time to understand the exploitability of the vulnerability
   ☐ Other

5. How much do you think DiffCVSS can reduce your workload?
   (scale question from reducing 0% - 100% workload)

6. What kinds of manual work still required with DiffCVSS?
   ☐ Look into the patch of the vulnerability
   ☐ Look into the CVE description
   ☐ Checking the reachability of the call-chain
   ☐ other

7. How precise do you think about DiffCVSS?
   (a) Extremely precise
   (b) Very precise
   (c) Moderately precise
   (d) Slightly precise

(e) Not precise at all

8. Will you consider using the tool in the future when you need to evaluate a new vulnerability?
   ☐ Yes
   ☐ No
   ☐ Might or might not

9. Do you think our methodology can be generalized to other programs/other applications?
   ☐ Yes
   ☐ No
   ☐ Might or might not

10. If any, please describe the shortcoming of DiffCVSS and anything that can be improved for DiffCVSS (open-question).

Program	**File**	**Line#**	**Impact**	**Category**
OpenSSL	crypto/x509/v3_crld.c	85	ML	D3
	crypto/cms/cms_sd.c	326	ML	D3
	crypto/store/loader_file.c	406	DF	D2
FreeBSD	lib/libkiconv/kiconv_sysctl.c	50	ML	D3
	lib/libkiconv/kiconv_sysctl.c	75	ML	D3

Table A.1: DiEH bugs found in OpenSSL and FreeBSD. D1, D2, D3 denote incorrect-order, redundant, and inadequate DiEH bugs, respectively. Column "Line#" is the line number, and Column 4 indicates impact of bug. ML = memory leak, DF = double-free.

Buggy func name	Imp	Cat.	S	R
add_mdev_supported_type	RL	D1	A	
dmi_sysfs_register_handle	RL	D3	S	SIQ
kfd_topology_update_sysfs	RL	D3	S	IQ
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
kfd_build_sysfs_node_entry	RL	D3	S	
fimc_md_register_sensor_entities	RL	D3	S	SIQ
NILFS_DEV_INT_GROUP_FNS	RL	D3	C	
power_supply_add_hwmon_sysfs	ML	D3	A	SIQ
intel_gtt_setup_scratch_page	ML	D3	A	IQ
nilfs_sysfs_create_snapshot_group	RL	D3	A	
acpi_cppc_processor_probe	RL	D3	A	SIQ
edac_device_register_sysfs_main_kobj	RL	D3	A	SIQ
netdev_queue_add_kobject	RL	D3	C	SIQ
nilfs_sysfs_create_snapshot_group	RL	D3	C	
bq24190_charger_get_property	RL	D3	S	SIQ
bq24190_charger_set_property	RL	D3	S	SIQ
bq24190_battery_get_property	RL	D3	S	SIQ
bq24190_battery_set_property	RL	D3	S	SIQ
stm32_mdma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dma_alloc_chan_resources	RL	D3	S	SIQ
tegra_adma_alloc_chan_resources	RL	D3	C	SIQ
stm32_dmamux_route_allocate	RL	D3	S	IQ

Table A.2: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).

Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R	Buggy func name	Imp	Cat.	S	R
aspeed_video_probe	ML	D3	C	SIQ	stm32f7_i2c_xfer	RL	D3	S	SIQ	rcar_pcie_probe	RL	D1	C	SIQ
nfp_abm_vnic_set_mac	ML	D3	A		stm32f7_i2c_reg_slave	RL	D3	S	SIQ	xcan_probe	RL	D1	S	SIQ
mlx4_opreq_action	-	D3	A		nv50_mstc_detect	RL	D3	C	SIQ	xcan_open	RL	D1	S	SIQ
rxkad_verify_response	ML	D3	A		nouveau_fbcon_open	RL	D3	S	SIQ	fec_enet_mdio_read	RL	D1	C	SIQ
siw_create_qp	ML	D3	C		nouveau_drm_ioctl	RL	D3	S	SIQ	macb_mdio_read	RL	D1	C	SIQ
cas_init_one	ML	D3	A	SIQ	radeon_drm_ioctl	RL	D3	C	SIQ	macb_mdio_write	RL	D1	C	SIQ
mlx4_opreq_action	ML	D3	A		radeon_crtc_set_config	RL	D3	C		omap4_keypad_probe	RL	D1	S	SIQ
add_port	ML	D3	A	SIQ	cdns_dsi_transfer	RL	D3	C	SIQ	mic_pre_enable	RL	D1	C	SIQ
img_i2s_in_probe	RL	D3	A	SIQ	v3d_get_param_ioctl	RL	D3	S	SI	img_spdif_in_probe	RL	D1	C	SIQ
iommu_group_alloc	RL	D3	A	SIQ	v3d_v3d_debugfs_ident	RL	D3	S	S	nouveau_drm_open	RL	D1	S	IQ
pblk_sysfs_init	RL	D3	C		v3d_measure_clock	RL	D3	S	S	radeon_dp_detect	RL	D1	S	
configfs_rmdir	RL	D3	C	SIQ	v3d_job_init	RL	D3	S		radeon_vga_detect	RL	D1	S	
f2fs_init_sysfs	RL	D3	C	SIQ	dss_runtime_get	RL	D3	S	SIQ	radeon_tv_detect	RL	D1	S	
f2fs_register_sysfs	RL	D3	C	S	dsi_runtime_get	RL	D3	S	SIQ	radeon_lvds_detect	RL	D1	S	
pci_create_slot	RL	D3	A	SIQ	venc_runtime_get	RL	D3	S	SIQ	bdisp_probe	RL	D1	S	SIQ
bond_sysfs_slave_add	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	bdisp_start_streaming	RL	D1	S	SIQ
iscsi_boot_create_kobj	RL	D3	A	SIQ	hdmi_runtime_get	RL	D3	S	SIQ	hva_hw_probe	RL	D1	C	SIQ
rx_queue_add_kobject	RL	D3	C	SIQ	dispc_runtime_get	RL	D3	S	SIQ	hva_hw_get_ip_version	RL	D1	S	SIQ
img_spdif_out_probe	RL	D3	C	SIQ	clk_pm_runtime_get	RL	D3	C	SIQ	coda_open	RL	D1	C	SIQ
rvt_create_qp	ML	D3	A		musb_irq_work	RL	D3	C	SIQ	fimc_is_probe	RL	D1	C	SIQ
gfs2_create_inode	RL	D3	C	SIQ	usb_port_resume	RL	D3	S	SIQ	fimc_lite_open	RL	D1	S	SIQ
ath10k_sta_state	RL	D3	C	SIQ	ina3221_write_enable	RL	D3	S	S	dcmi_start_streaming	RL	D1	S	SIQ
ccp_run_sha_cmd	ML	D3	C		gpmi_nfc_exec_op	RL	D3	C	SIQ	s3c_camif_probe	RL	D1	C	SIQ
rockchip_pdm_resume	RL	D3	A	SIQ	bch_set_geometry	RL	D3	C	SIQ	img_i2s_in_probe	RL	D1	C	SIQ
tegra30_ahub_resume	RL	D3	A	SIQ	delta_get_sync	RL	D3	S	SIQ	venc_open	RL	D1	C	SIQ
tegra30_i2s_resume	RL	D3	A	SIQ	hva_hw_dump_regs	RL	D3	S	S	vfe_get	RL	D1	S	SIQ
img_i2s_out_set_fmt	RL	D3	C	I	stm32f7_i2c_reg_slave	RL	D3	S	SIQ	exynos_trng_probe	RL	D1	C	SIQ
img_i2c_xfer	RL	D3	C	SIQ	isp_video_open	RL	D3	C	SIQ	rvin_open	RL	D1	C	SIQ
configfs_rmdir	RL	D3	C	SIQ	s5pcsis_s_stream	RL	D3	S	SIQ	rvt_create_qp	ML	D1	A	
img_prl_out_set_fmt	RL	D3	A	I	fimc_capture_open	RL	D3	C	SIQ	rawsock_connect	RL	D1	S	SIQ
ethoc_probe	ML	D3	S	SIQ	vpe_runtime_get	RL	D3	S	SIQ	lpi2c_imx_master_enable	RL	D3	S	SIQ
img_i2s_out_probe	RL	D3	C	SIQ	xiic_xfer	RL	D3	S	SIQ	panfrost_job_hw_submit	RL	D3	S	SIQ
img_i2c_init	RL	D3	C	SIQ	s3c_camif_open	RL	D3	C	SIQ	vc4_dsi_encoder_enable	RL	D3	S	SIQ
img_i2c_xfer	RL	D3	C	SIQ	s5p_mfc_power_on	RL	D3	S	SIQ	vc4_vec_encoder_enable	RL	D3	S	SIQ
display_init_sysfs	RL	D3	A	SIQ	img_i2s_in_set_fmt	RL	D3	C	I	cpuidle_add_state_sysfs	RL	D3	A	IQ
bq24190_sysfs_show	RL	D3	S	SIQ	csid_set_power	RL	D3	C	SIQ	efivar_create_sysfs_entry	RL	D3	C	SIQ
bq24190_sysfs_store	RL	D3	S	S	ispif_set_power	RL	D3	C	SIQ	esre_create_sysfs_entry	RL	D3	A	SIQ
img_pwm_remove	RL	D3	S	SIQ	csiphy_set_power	RL	D3	S	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
img_pwm_config	RL	D3	S	SIQ	vsp1_probe	RL	D3	C	SIQ	dwc3_pci_resume_work	RL	D3	C	SIQ
ti_qspi_setup	RL	D3	C	SIQ	rcar_fcp_enable	RL	D3	S	SIQ	cdns_dsi_bridge_enable	RL	D3	C	SIQ
tegra_sflash_resume	RL	D3	C	SIQ	__vxlan_dev_create	ML	D3	C	Q	nfc_genl_llc_set_params	UAF	D2	C	IQ
tegra_spi_setup	RL	D3	S	SIQ	cpuidle_add_sysfs	RL	D3	A	SIQ	wlcore_regdomain_config	RL	D3	C	IQ
tegra_spi_resume	RL	D3	S	SIQ	fw_cfg_register_file	RL	D3	S	SIQ	radeon_driver_open_kms	RL	D3	C	
sprd_spi_remove	RL	D3	C	SIQ	edd_device_register	RL	D3	S	SIQ	nouveau_crtc_set_config	RL	D3	-	SIQ
tegra_slink_setup	RL	D3	C	SIQ	dmi_system_event_log	RL	D3	S	IQ	rga_buf_start_streaming	RL	D3	C	
tegra_slink_resume	RL	D3	C	SIQ	mc13xxx_rtc_probe	DU	D2	A	SIQ	s5p_jpeg_start_streaming	RL	D3	S	SIQ
img_spfi_resume	RL	D3	S	SIQ	m66592_probe	DF	D2	A	SIQ	stm32f7_i2c_smbus_xfer	RL	D3	S	SIQ
edma_probe	RL	D3	C	SIQ	cros_ec_ishtp_probe	DU	D2	S	SIQ	mtk_jpeg_start_streaming	RL	D3	S	SIQ
rcar_dmac_probe	RL	D3	S	SIQ	punch_hole	DU	D2	-	SIQ	stm32f7_i2c_unreg_slave	RL	D3	S	SIQ
sprd_dma_remove	RL	D3	S	SIQ	nfc_genl_llc_sdreq	UAF	D2	C	IQ	fimc_isp_subdev_s_power	RL	D3	S	SIQ
zpa2326_resume	RL	D3	C	SIQ	qcom_pcie_probe	-	D2	S	SIQ	nouveau_gem_object_del	RL	D3	S	
arizona_clk32k_enable	RL	D3	A	SIQ	s3c_camif_probe	-	D1	A	SIQ	panfrost_perfcnt_enable_locked	RL	D3	S	
gpio_rcar_request	RL	D3	C	SIQ	tegra_adma_probe	-	D1	C	SIQ	etnaviv_gpu_recover_hang	RL	D3	S	SIQ
arizona_gpio_get	RL	D3	A	SIQ	i915_gem_init	ML	D1	C	IQ	arizona_gpio_direction_out	RL	D3	A	SIQ
sata_rcar_resume	RL	D3	A	SIQ	pvrdma_pci_probe	-	D1	A	SIQ	vc4_hdmi_encoder_enable	RL	D3	S	SIQ
sata_rcar_restore	RL	D3	A	SIQ	qib_create_port_files	RL	D1	A		amdgpu_display_crtc_set_config	RL	D3	S	
cdns_pcie_host_probe	RL	D3	S	SIQ	add_port	RL	D1	A	SIQ	nouveau_connector_detect	RL	D3	S	SIQ
cdns_pcie_ep_probe	RL	D3	-	SIQ	i915_gem_init	ML	D1	-	IQ	nv50_disp_atomic_commit	RL	D3	C	SIQ
xcan_get_berr_counter	RL	D3	S	SIQ	test_hints_case	RL	D1	A	SIQ	edac_pci_main_kobj_setup	RL	D3	A	IQ
fec_enet_open	RL	D3	C	SIQ	gfs2_create_inode	RL	D1	C	SIQ	nouveau_gem_object_open	RL	D3	C	
fec_enet_mdio_write	RL	D3	C	SIQ	rocker_dma_rings_init	ML	D1	A	SIQ	nouveau_debugfs_pstate_set	RL	D3	C	SIQ
bma150_open	RL	D3	S	SIQ	tegra_spi_probe	RL	D1	S	SIQ	nouveau_debugfs_strap_peek	RL	D3	S	S
stmfts_input_open	RL	D3	S	IQ	tegra_slink_probe	RL	D1	C	SIQ	amdgpu_connector_dp_detect	RL	D1	S	
stm32f7_i2c_xfer	RL	D3	S	SIQ	tegra_adma_probe	RL	D1	C	SIQ	amdgpu_connector_vga_detect	RL	D1	S	
arizona_extcon_probe	RL	D3	C	SIQ	usb_dmac_probe	RL	D1	S	SIQ	amdgpu_connector_lvds_detect	RL	D1	S	
etnaviv_gpu_init	RL	D3	S	SIQ	sprd_dma_probe	RL	D1	S	SIQ	amdgpu_driver_open_kms	RL	D1	S	
etnaviv_gpu_debugfs	RL	D3	S		sata_rcar_probe	RL	D1	A	SIQ	tegra_vde_ioctl_decode_h264	RL	D1	C	SI
etnaviv_gpu_bind	RL	D3	S	SIQ	tegra_pcie_probe	RL	D1	S	SIQ	qlcnic_83xx_interrupt_test	ML	D1	A	I
vc4_v3d_pm_get	RL	D3	S	SI	qcom_pcie_probe	RL	D1	S	SIQ	acpi_sysfs_add_hotplug_profile	RL	D1	A	IQ
amdgpu_drm_ioctl	RL	D3	S	SIQ	dra7xx_pcie_probe	RL	D1	S	SIQ	nilfs_sysfs_create_device_group	RL	D1	C	S

Table A.3: Summary of DiEH bugs detected by HERO in Linux kernel v5.3. Column(Col) 1 denotes functions containing DiEH bug. Col 2 (Imp) indicates the impact of the bug. ML = memory leak, UAF = use-after-free/double-free, DU = double-unlock, RL = refcount leak. Col 3 (Cat.) indicates the category of DiEH bugs with D1 = incorrect order, D2 = redundant, D3 = inadequate follower function. Col 4 (S) indicates the status of the patch with S, A, C, and - indicating submitted, accepted, confirmed, and file not existing in the latest version, respectively. Col 5 (R) indicates the bug's reachability from system calls (S), I/O control handlers (I), and IRQ handlers (Q).

Git commit	BT	SI	ST	CVSS	ET	Git commit	BT	SI	ST	CVSS	ET	Git commit	BT	SI	ST	CVSS	ET
bf703c3f19934	MBC	OBA	N			f4351a199cc12	MBC	OBA	R/U	7.2	SII	3af54c9bd9e6f	MI	UU	N	1.9	SC
74415a36767d9	MBC	OBA	N			d7ac3c6ef5d8c	MBC	OBA	R/U	4.7	SII	97e69aa62f8b5	MI	UU	N	1.9	IO
9ed87fd34c97a	MBC	OBA	N		SII	04f25edb48c44	MBC	OBA	R/U	7.2	SII	c4c896e1471ae	MI	UU	N	1.9	
494264379d186	MBC	OBA	N		SII	5d6751eaff672	MBC	OBA	R/U	9.4	SII	8d03e971cf403	MI	UU	N	1.9	
3d0ccd021b23c	MBC	OBA	N			de591dacf3034	MI	UU	N		SII	792039c73cf17	MI	UU	N	1.9	
ec3cbb9ce241d	MBC	OBA	R			325fb5b4d2603	MI	UU	N		SII/F	9344a972961d1	MI	UU	N	1.9	
82033bc52abeb	MBC	OBA	N		SII	2fc2111c27294	MI	UU	R/D		IO	e862f1a9b7df4	MI	UU	N	1.9	
b56fa1ed09615	MBC	OBA	N		SII	ed77ed6112f2d	MI	UU	R/D			1f86840f89771	MI	UU	N	1.9	SII
62c8ba7c58e41	MBC	OBA	N		IO	ce384d91cd7a4	MI	UU	R/D		IO	c88e739b1fad6	MI	UU	N	4.3	
5d60122b7e30f	MBC	OBA	R		SII	1a8b7a67224eb	MI	UU	R/D		SII	96b340406724d	MI	UU	N	1.7	
a9ae4692eda4b	MBC	OBA	R		SII	b5f15ac4f89f8	MI	UU	R/D		SII	8e3fbf870481e	MI	UU	N	1.9	
b9f62ffe05e40	MBC	OBA	R		SII	cccbe5ef85284	MI	UU	R/D		SII	b6878d9e03043	MI	UU	N	2.1	IO
d3c2155ce5889	MBC	OBA	R		SII	eca67aaeebd6e	MI	UU	R/D		IO	eda98796aff0d	MI	UU	N	1.9	
12f4543f5d681	MBC	OBA	R		SII	a0c5a3944ce12	MI	UU	R/D			681fef8380eb8	MI	UU	N	2.1	IO
9a07826f99034	MBC	OBA	R		SII	5b919f833d9d6	MI	UU	R/D			342ffc26693b5	MI	UU	R	2.1	IO
e1718d97aa88e	MBC	OBA	R		SII/F	938abd8449c27	MI	UU	R/D			0625b4ba1a5d4	MI	UU	R/U	2.1	SII
49521b13cbc02	MBC	OBA	R		SII	144ce879b057c	MI	UU	N			3b7d2b319db0b	MN	DF	N		SII
ef6ff8f47263b	MBC	OBA	R		SII	99b0d365e5ade	MI	UU	N		SII	1cfafab965198	MN	DF	R		SII
f716abd55d1e1	MBC	OBA	R		SII	9561f7faa45cb	MI	UU	R/D		IO	266e8ae37daa0	MN	DF	R		
8986a11978373	MBC	OBA	R		SII	21dba24481f70	MI	UU	R/D		SII	f5c4441cd8012	MN	DF	R		SII
f658f17b5e0e3	MBC	OBA	R		SII	e7332691de2f9	MI	UU	R/D			1c963bec3534b	MN	DF	N		SII
952e5daa2565f	MBC	OBA	U/R/D			02745f63443c0	MI	UU	R/D			1eb8f7a7da6d3	MN	DF	R		SII
8513027a73c2f	MBC	OBA	N			47966e9779528	MI	UU	R/D		IO	1adb2e2b5f850	MN	DF	R		
3a63e44420932	MBC	OBA	R/D		SII	5ffedc6ed3d06	MI	UU	R/D			f69ae770e74df	MN	DF	N		SII
b481077354fe	MBC	OBA	R/D			5dbd5068430b8	MI	UU	R			eb1716af88737	MN	DF	N		
ac57245215696	MBC	OBA	R/D		SII	282c4c0ecce9b	MI	UU	R/D		IO	c654ecbbfefbe	MN	DF	R		SII
6e893ca25e9ea	MBC	OBA	R/D		SII	dc43376c26cef	MI	UU	R/D			6dd93e9e5eb19	MN	DF	R		SII
1e7eb89ba936f	MBC	OBA	N		SII	b08e1ed9cfcf7	MI	UU	R/D		SII	1e7bac1ef754b	MN	DF	R		SII
5270041d342de	MBC	OBA	N		SII	b09c74ae1263e	MI	UU	R/D		SII	6cae6d3189ef3	MN	DF	R		
32fb5f06dbb6c	MBC	OBA	N		SII	6ea437a3639b1	MI	UU	R/D		SII	b1214e4757b7d	MN	DF	R		SII
42d8644bd77dd	MBC	OBA,	R		SII	d14df339c72b6	MI	UU	R/D			32b85442961b94	MN	DF	R		F
		NPD				bffbbc0a2ccb9	MI	UU	R/D		IO	7d78874273463	MN	DF	R		SII
a0a74e45057cc	MBC	OBA,	N		SII	3ca9e5d36afb5	MI	UU	R/D		IO	c1b03ab5e8867	MN	DF	R		SII
		NPD				ee7ff5fed2571	MI	UU	R/D		SII	d7426c69a1942	MN	DF	R		SII/F
8dd2c9e3128a5	MBC	OBA,	N			c9889803e3ba6	MI	UU	R/D		SII	b3b51417d0af6	MN	DF	R		
		NPD				7e8631e8b9d4e	MI	UU	R/D		SII	f683c80ca68e0	MN	DF	R		SII/F
518ff04fd8429	MBC	OBA,	N		SII	a5f6fc28d6e6c	MI	UU	R/D			7dc4a6b5ca942	MN	DF	R		SII
		NPD				81907478c4311	MI	UU	R/D		SII	23418dc131464	MN	DF	R/U/D		
bb1553c800227	MBC	OBA,	N		SII	a44b0f5edfc63	MI	UU	R/D			7fafcfdf6377b	MN	DF	R		SII
		NPD				5899f0478528b	MI	UU	R/D		SII	ef2a7cf1d8831	MN	DF	R/U		SII/F
55e8dba1acc2e	MBC	OBA,	R			12b055662ac62	MI	UU	R/D		SII	dc035d4e934e5	MN	DF	R		SII
		NPD				79b568b9d0c7c	MI	UU	R/D		SII	4d45e21867bee	MN	UAF	N		SII
60ad768933ec1	MBC	OBA,	N		SII	aee177ac5a422	MI	UU	R/D		SII	f3429545d03a5	MN	UAF	N		
		NPD				62d494ca27735	MI	UU	R/D		SII	e04ca626baee6	MN	UAF	N		
0b1d250afb8eb	MBC	OBA,	N		SII	ab73ef46398e2	MI	UU	R/D		SII	e3e14de50dff8	MN	UAF	N		
		NPD				0b857b44b5e44	MI	UU	N		SII	3fc98b1ac0366	MN	UAF	N		SII
eef08e5350618	MBC	OBA,	N		SII	9cd70e80f7f0d	MI	UU	R/D			8f68ed9728193	MN	UAF	N		
		NPD				7307616245bab	MI	UU	N			6cf9e995f91e5	MN	UAF	N		SII
221be106d75c1	MBC	OBA,	R/U		SII	02a9079c66341	MI	UU	R/D			ba54238552625	MN	UAF	R		SII/F
		NPD				d69bb92e402ff	MI	UU	R/D		IO	715252d419129	MN	UAF	N		SII
ef4b4856593fc	MBC	OBA,	R		SII	0f4bbb233743b	MI	UU	R/D		IO	ece1d77ed73b3	MN	UAF	N		SII
		NPD				5b0907407e7f2	MI	UU	R/D		SII	f276795627045	MN	UAF	R		SII
2a2f11c227bdf	MBC	OBA	R	7.5	SII	e15882b6c6caf	MI	UU	R/D		SII/F	0d012b9866249	MN	UAF	R		
0031c41be5c52	MBC	OBA	R	7.5	SII	72ccc471e13b8	MI	UU	R/D		SII/F	e8243f32f2550	MN	UAF	R		SII
0926f91083f34	MBC	OBA	R	7.5	SII	b51456a6096eb	MI	UU	R/D			fc09149df6e20	MN	UAF	R		
12f09ccb46127	MBC	OBA	R	7.5		f7a6cb7b38c68	MI	UU	R/D			29322d0db98e5	MN	UAF	R		
2da424b0773ce	MBC	OBA	R	7.5	SII/F	6ce14f6416c84	MI	UU	R/D		SII/F	400ffaa2acd72	MN	UAF	R		SII
43622021d2e2b	MBC	OBA	N	6.2	SII/F	4c5009c5256d0	MI	UU	R/D		SII	44aa91ab2bb86	MN	UAF	R		SII/F
78214e81a1bf4	MBC	OBA	N	4.7	SII/F	df7e40425813c	MI	UU	R/U			a723bab3d7529	MN	UAF	R		SII
41df7f6d43723	MBC	OBA	N	4.7	F	d0ea2b1250054	MI	UU	R/U		SII/F	c7de572630762	MN	UAF	R		SII
0fb6bd06e0679	MBC	OBA	N	4.7	SII/F	2d93913e22013	MI	UU	R/D			fa6114d4bde70	MN	UAF	R		SII
297502abb32e2	MBC	OBA	N	5.4	SII/F	5540fbf438458	MI	UU	R/U			a7a7aeefbca29	MN	UAF	R		IO/F
		NPD				40f7090bb1b4e	MI	UU	R/U			c5540a0195ec6	MN	UAF	R		
1fa2337a315a2	MBC	OBA	R	7.5	SII	81114baa835b5	MI	UU	R/U		SII	8667f515952fe	MN	UAF	R		SII
9d47964bfd471	MBC	OBA	R	4.6		58796e67d5d52	MI	UU	R/U			741b8b832a574	MN	UAF	R		
193e87143c290	MBC	OBA	R	7.5	SII	6d084ac27ab4b	MI	UU	R/U		SII	031e5896dfdc2	MN	UAF	R		
780e982905bef	MBC	OBA	R	4.6		e4818d615b58f	MI	UU	N		SII	3587cb87cc44c	MN	UAF	R		
b550a32e60a49	MBC	OBA	N	7.8		e55449e71aade	MI	UU	N		F	87fc030231b11	MN	UAF	R		SII
		NPD				84ce4d0f9f55b	MI	UU	N		SII	5bfd37b4de5c9	MN	UAF	R		SII/F
dad5ab0db8dea	MBC	OBA	N	7.2	SII	354d0fab649d4	MI	UU	R/U		SII	6818caa4cdc95	MN	UAF	N		SII/F
		NPD				982f7c2b2e6a2	MI	UU	N	1.9	SC	0739310117b49d	MPC	PE	N	3.6	SII
db7683d7deb25	MN	UAF	R			abd39c6ded9db	MN	UAF	R/U	4.9	SII	497de07d89c14	MPC	PE	N	3.6	F
6d6340672ba3a	MN	UAF	R		SII/F	4397f04575c44	MN	DF	R/U	7.2	SII	bfc81a8bc18e3	MBC	OBA	N	7.2	SII/F
25524288631fc	MN	UAF	R			d024206133ce2	MPC	PE	R			0c319d3a144d4	MBC	OBA	R	7.5	SII
e9777ad4399c2	MN	UAF	R/U/D			d2c2b11cfa134	MPC	PE	R			89c6efa61f570	MBC	OBA	R	4.6	SII/F
11e40f5c57762	MN	UAF	R/U		SII/F	ed82571b1a14a	MPC	PE	R			6acb47d1a318e	MBC	OBA	R	4.6	SII
ae6ccb0f8153f	MN	UAF	R		SII	f2b20f6ee8423	MPC	PE	R/D			4da62fc70d7cb	I	UU,	R/D		
c278c253f3d99	MN	UAF	R	4.4	SII/F	e3a2b93ddddad3	MPC	PE	R		SII			PE,			
36e4ad0316c01	MN	UAF	R	6.1	SII	03ce7b1d23498	MPC	PE	R/U		SII			OBA			
c9fbd7bbc23db	MN	UAF	R/U	4.6	SII	41bdc78544b8a	MPC	PE	N	2.1	SII	49c37c0334a9b	I	UU	N	4.9	
54648cf1ec2d7	MN	UAF	R/U	4.6		c0ca3d70e8d3c	MPC	PE	N	4.9		fd02db9de73fa	I	UU	N	1.9	IO

Table A.4: List of security bugs detected by SID. BT: bug type; SI: security impact; ET: entry point types; MBC: missing/wrong bound check; OBA: out-of-bound access; MI: missing initialization; NPD: NULL-pointer dereference; UU: uninitialized use; MN: missing nullification; MPC: missing permission check; DF: double-free; UAF: use-after-free; PE: permission bypass; U: unpatched bug in Android 4.14-p; R: requested CVE ID; D: CVE request declined by CVE maintainers; N: not request CVE ID; SC: system-call; IRQ: IRQ handlers; IO: I/O control handlers; SII: system-call & I/O control handler & IRQ handlers; F: confirmed dynamically (by Fuzzer); CVSS: common vulnerability scoring system.

Security bugs (by root cause)	**Security impact**	**Security operation**	**Vulnerable operation**	**Critical Variable**	**SR in PV or SO in UPV**	**SR in UPV or SO in PV**
Missing release	Memory leak	Release operation	Allocation operation	Allocated pointer	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
Missing NULL check	NULL dereference	NULL check	Pointer dereference	Checked pointer	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
Missing zero check	Divide by zero	Zero check	Use as divisor	Divisor	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$
Missing/wrong locks/unlocks	Race condition (UAF/DF and etc.)	Lock/unlock	Operations in critical section	Lock variable	$\text{FLAG}_{CV} = 0$	$\text{FLAG}_{CV} = 1$

Table A.5: The key components of patches and constraints modeling for more types of bugs. SO = security operations; SR = security rules; PV = patched version; UPV = unpatched version.

System/ # of targets	<10	10-100	100-1000	>= 1000	TotalTargets	Total Icalls	Mean	MAX
Linux (Type-based)	77.0%	15.8%	5.4%	1.8%	4734762	55921	84.7	8862
Linux (GNNIC, Sim = 0.9)	85.7%	13.0%	1.3%	0.0%	545452	55921	9.7	2436
Android (Type-based)	82.8%	10.0%	5.6%	1.6%	4769716	62618	76.1	9056
Android (GNNIC, Sim = 0.9)	94.9%	4.5%	0.6%	0.0%	297284	62618	4.7	2645
FreeBSD (Type-based)	85.5%	11.7%	1.3%	1.5%	251307	7578	33.2	1960
FreeBSD (GNNIC, Sim = 0.9)	88.5%	11.3%	0.2%	0.0%	34669	7578	4.5	217

Table A.6: Distribution of indirect calls that have a number of targets in the range (specified on the first row). Type-based: Original type-based approach on the system; Sim=0.9: The results of GNNIC based on the similarity at 0.9.

Table A.7: Misused "allocation" related functions in the Linux and FreeBSD kernels. ITT = identified total used times; IMT = identified misused times (bugs); F = flag to indicate if the cross-checking-only approaches can find the issue; Linux kernel is at git commit dbd736c8116f; FreeBSD kernel is at git commit 4ba619efbdd2476;

Target	Misused function name	ITT	IMT	F
Linux	proc_net_mkdir	7	1	Y
Linux	nci_skb_alloc	9	1	Y
Linux	bio_alloc_bioset	5	1	Y
Linux	kcalloc_node	14	1	Y
Linux	dma_alloc_attrs	19	1	Y
Linux	alloc_wrb_handle	4	1	Y
Linux	cdns3_gadget_ep_alloc_request	2	1	N
Linux	v4l2_ctrl_new_custom	23	4	Y
Linux	drm_atomic_get_new_connector_state	6	1	Y
Linux	acpi_os_allocate	5	2	Y
Linux	acpi_os_allocate_zeroed	7	1	Y
Linux	acpi_os_allocate_zeroed	3	1	Y
Linux	nfp_cpp_mutex_alloc	3	1	Y
Linux	alloc_irq_cpu_rmap	9	1	Y
Linux	pblk_alloc_rqd	4	2	N
Linux	__alloc_object	2	1	N
Linux	alloc_buffer_head	2	1	N
Linux	alloc_page_buffers	3	1	Y
Linux	kcalloc_node	14	1	Y
Linux	nci_skb_alloc	9	1	Y
Linux	alloc_pages_node	12	3	Y
Linux	v4l2_ctrl_new_int_menu	17	1	Y
Linux	i2c_new_device	9	1	Y
Linux	v4l2_ctrl_new_std_menu	32	2	Y
Linux	compat_alloc_user_space	15	15	N
Linux	iova_magazine_alloc	3	3	N
Linux	nd_dax_alloc	2	2	N
Linux	usbhs_pipe_malloc	2	2	N
FreeBSD	xpt_alloc_ccb	8	3	Y
FreeBSD	cam_simq_alloc	27	1	Y
FreeBSD	nvme_allocate_request_vaddr	10	5	N
FreeBSD	g_malloc	31	13	Y
FreeBSD	devfs_alloc	3	1	Y
FreeBSD	if_alloc	69	3	Y
FreeBSD	bit_alloc	2	1	N
FreeBSD	sglist_alloc	12	1	Y
FreeBSD	buf_ring_alloc	4	1	Y
FreeBSD	uma_zalloc	64	14	Y